JavaScript+Vue +React 全程实例



通过JavaScript实例掌握Web前端开发技术

- 涵盖目前流行的特效、流行的JS技术、流行的Vue与React框架
- 包括众多JavaScript应用场景，直接感受实际开发出来的页面效果
- 围绕实例进行讲解，每一节都可以让读者掌握一种实用技术

郑均辉　薛 燚　编著


范例程序源代码

清华大学出版社

JavaScript+Vue +React 全程实例



郑均辉　薛 燚　编著

清華大学出版社
北 京

内 容 简 介

本书基于理论知识与开发实践相结合的思想，精选当前简单、实用和流行的百余个 JavaScript 代码实例，帮助读者学习掌握 JavaScript 脚本语言。全书内容翔实、重点突出、通俗易懂，涵盖了 JavaScript 前端开发的方方面面。

全书共分为 13 章，包括 JavaScript 前端设计、调试和开发的一些必备知识，表单处理、DOM 控制、控件特效、日期时间、网页特效、DIV+CSS、Ajax 应用等方面的应用实例，还特别增加了对当下非常流行的 React 和 Vue.js 框架的介绍。本书的全部代码实例均是对 JavaScript 技术最具代表性的实践应用，可以帮助读者深入学习 JavaScript 的开发技巧。

本书是学习掌握 JavaScript 技术非常好的图书，既适合 JavaScript、Vue、React 前端初学者阅读，也适合从事前端网页设计以及需要学习前端技术的后端开发工程师阅读，同时还可作为高等院校和培训学校相关专业的教材。相信本书丰富的内容和大量的实例能够帮助初学者快速步入 Web 前端开发的捷径，并衷心地希望每一名前端爱好者都可以成为有代码实践和技术深度的 JavaScript 高手。

图书在版编目（CIP）数据

JavaScript+Vue+React 全程实例/郑均辉，薛燚编著．—北京：清华大学出版社，2019
（Web 前端技术丛书）
ISBN 978-7-302-53164-7

Ⅰ．①J…　Ⅱ．①郑…　②薛…　Ⅲ．①JAVA 语言－程序设计　Ⅳ．①TP312.8

中国版本图书馆 CIP 数据核字（2019）第 116023 号

责任编辑：夏毓彦
封面设计：王　翔
责任校对：闫秀华
责任印制：杨　艳

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　**邮　　编：**100084
社 总 机：010-62770175　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：清华大学印刷厂
经　　销：全国新华书店
开　　本：190mm×260mm　**印　　张：**20.5　**字　　数：**525 千字
版　　次：2019 年 8 月第 1 版　**印　　次：**2019 年 8 月第 1 次印刷
定　　价：69.00 元

产品编号：082162-01

前　　言

读懂本书

JavaScript 无处不在

二十多年前，布兰登•艾奇（Brendan Eich）为 Netscape 浏览器草草地设计出网页脚本语言（最早的 JavaScript 原型）的时候，可能根本不会预料到如今 JavaScript 会成为 Web 开发领域的第一编程语言。

在权威的编程语言排行榜（TIOBE）中，JavaScript 多年来一直稳居在前几名之列。尽管自 JavaScript 诞生之日起，就伴随着诸如“语法不够严谨”“逻辑不够清晰”“代码管理混乱”这类的批评之声，但这并没有阻止广大程序员对其的喜爱。

JavaScript 之所以能够得到广泛的欢迎和普及，与其简单易学、使用灵活，跨平台兼容的这些特性密不可分。新手可以很快地掌握一些基本技巧并实践在网页开发中，高手可以凭借扎实的基本功、构建出逻辑复杂且功能强大的 Web 应用。

如今，JavaScript 支持在绝大多数的平台上进行开发：PC 客户端的应用程序，Web 服务器端的业务逻辑，嵌入式芯片设计，物联网设备研发等，均是 JavaScript 可以发挥魔力的舞台。毫不夸张地讲，今天的 JavaScript 几乎是无处不在。

JavaScript 技术特点

JavaScript 是一种基于对象（Object）和事件驱动（Event Driven）并具有相对安全性的脚本语言。JavaScript 广泛应用于互联网的 Web 开发，通过为 HTML 网页添加动态响应功能来提高用户交互体验。

JavaScript 的设计目标，就是一种基于原型对象、弱类型、事件驱动、跨平台兼容的解释性动态脚本语言。同时，由于 JavaScript 具有很强的扩展性，因此可以基于 JavaScript 原生语法开发出功能更为强大的类库或框架。

当然，也正是由于 JavaScript 的灵活性太强，各大浏览器厂商设计的版本兼容性不好。不过，这种情况随着 EMCA TC39 委员会推出的标准化 ECMAScript 脚本语言得到了很好的改善，相信未来 JavaScript 脚本语言的兼容性会越来越好。

JavaScript 扩展类库和框架

JavaScript 之所以无处不在、广受欢迎，相信与其庞大的扩展类库和框架群不无关系。例如，ProtoType、jQuery、jQuery Mobile、AngularJS、React、Vue.js 等，这些耳熟能详的名字都是 JavaScript 扩展类库和框架的优秀代表。

以上这些优秀的 JavaScript 扩展类库和框架不单单是将核心功能进行抽象、集成和扩展，而是在设计模式、功能架构、性能优化等方面做足了功夫，带给了设计人员无与伦比的编程体验以及代码性能和运行效率的显著提升。

本书真的适合你吗

本书大量的基础代码实例可以帮助读者快速掌握 JavaScript 的编程技巧，并应用到实践开发之中。尤其是关于 JavaScript 框架的提高内容中，通过对目前流行的 React 和 Vue.js 框架的介绍，帮助读者去了解前端 Web 技术的前沿方向。无论是基础内容或提高内容，相信读者都可以从中获益。

本书涉及的主要软件工具、技术或框架

Apache HTTP	Mozilla Firefox	Dreamweaver CS6	UltraEdit
WebStorm	Sublime Text	Notepad	EditPlus
HTML	MIME	HTTP	React
HTML5	JavaScript	ECMAScript	Vue.js
CSS3	AJAX	RegExp	JSON

本书特点

（1）本书完全是从简单、通用的 JavaScript 代码实例出发，抛开枯燥的纯理论知识介绍，通过实例讲解的方式帮助读者学习 JavaScript 脚本语言设计。

（2）本书内容涵盖 JavaScript 所涉及的、绝大部分的前端开发知识，将这些内容整合到一起可以系统地了解掌握这门语言的全貌，为介入大型 Web 项目的开发做了很好的铺垫。

（3）本书对于实例中的知识难点做出了详细的分析，能够帮助读者有针对性地提高 JavaScript 编程开发技巧。

（4）本书在知识点上按照类别进行合理的划分，全部的代码实例都是独立的，读者可以从头开始阅读，也可以从中间开始阅读，不会影响学习进度。

（5）本书代码遵循重构原理，避免代码污染，真心希望读者能写出优秀、简洁、可维护的代码。

代码下载

本书代码下载地址（注意数字与字母大小写）：https://share.weiyun.com/5K9SHun。如果下载有问题，请联系 booksaga@163.com，邮件主题为“JavaScript+Vue+React”。

本书读者

- JavaScript、Vue、React 前端开发初学者
- 从事前端网页设计的开发工程师
- 需要学习前端技术的后端开发工程师
- 高等院校和培训学校相关专业的师生

本书第 1~11 章由平顶山学院的郑均辉编写，第 12~13 章由薛燚编写。

关于封面照片

封面照片由蜂鸟网摄影家 ptwkzj 先生友情提供，在此表示衷心感谢。

编　者

2019 年 6 月

目　　录

第 1 章　JavaScript 环境搭建

本书将用新颖的视角去认识 JavaScript，通过简单流行的代码实例深度阐述 JavaScript 的特性，尽量利用 IT 世界里有意思的东西来激发读者的学习兴趣。本章将概括性地介绍 JavaScript 的书写方式、调试方式和开发工具。

1.1　HTML 中书写 JavaScript 的几种方式

编写 JavaScript 代码其实无须特殊软件，一个普通的文本编辑器（如记事本）和一个 Web 浏览器就足够了。用 JavaScript 编写的代码需要放在 HTML 文档中才能被浏览器执行，有以下两种方式可以做到这一点。

第一种方式是将 JavaScript 代码放到文档<head>标签的<script>标签中：

```
<!DOCTYPE html>
<html>
 <head>
  <title>hello world</title>
  <script>
     alert('hello world!');
  </script>
 </head>
 <body>
 </body>
</html>
```

将上面的代码保存到 HTML 文件中（在记事本中编写，然后另存为扩展名为 html 的文件），用任意浏览器打开，就可以看到一个弹出对话框。

第二种方式是把 JavaScript 代码存为一个扩展名为 js 的独立文件。以前的做法是在文档<head>里用<script>标签的 src 属性来指向该文件：

```
<!DOCTYPE html>
<html>
 <head>
  <title>hello world</title>
  <script src="helloworld.js"></script>
 </head>
 <body>
```

```
08   </body>
09  </html>
```

目前业界推荐的做法是把<script>放到 HTML 文档最后，</body>标签之前（第 08 行和第 09 行之间）。这样做的目的是，使浏览器更快地加载页面并展示给用户，从而增强用户体验。

1.2 选择开发工具

近几年 JavaScript 的开发工具也得到了蓬勃发展，大小工具琳琅满目，难易程度也是不尽相同。比如重量级的集成开发平台，包括 Adobe Dreamweaver、Visual Studio 系列和 jetBrains WebStorm 系列，具有简单易学、适应性强、功能完善等特点，深受广大开发者的喜爱。一些轻量级的代码编辑器工具（EditPlus 和 Sublime Text）也非常受专业人士欢迎，这些编辑器看似功能简单实则扩展性很强大，初学者虽不易掌握，但熟练运用后一定会让设计人员爱不释手。这里简单介绍三种 JavaScript 开发工具。

1. Adobe Dreamweaver

这就是曾经被誉为“网页三剑客”之一的 Dreamweaver，备受广大网页设计和开发人员的喜爱，历史非常悠久。它的运行效果如图 1.1 所示。

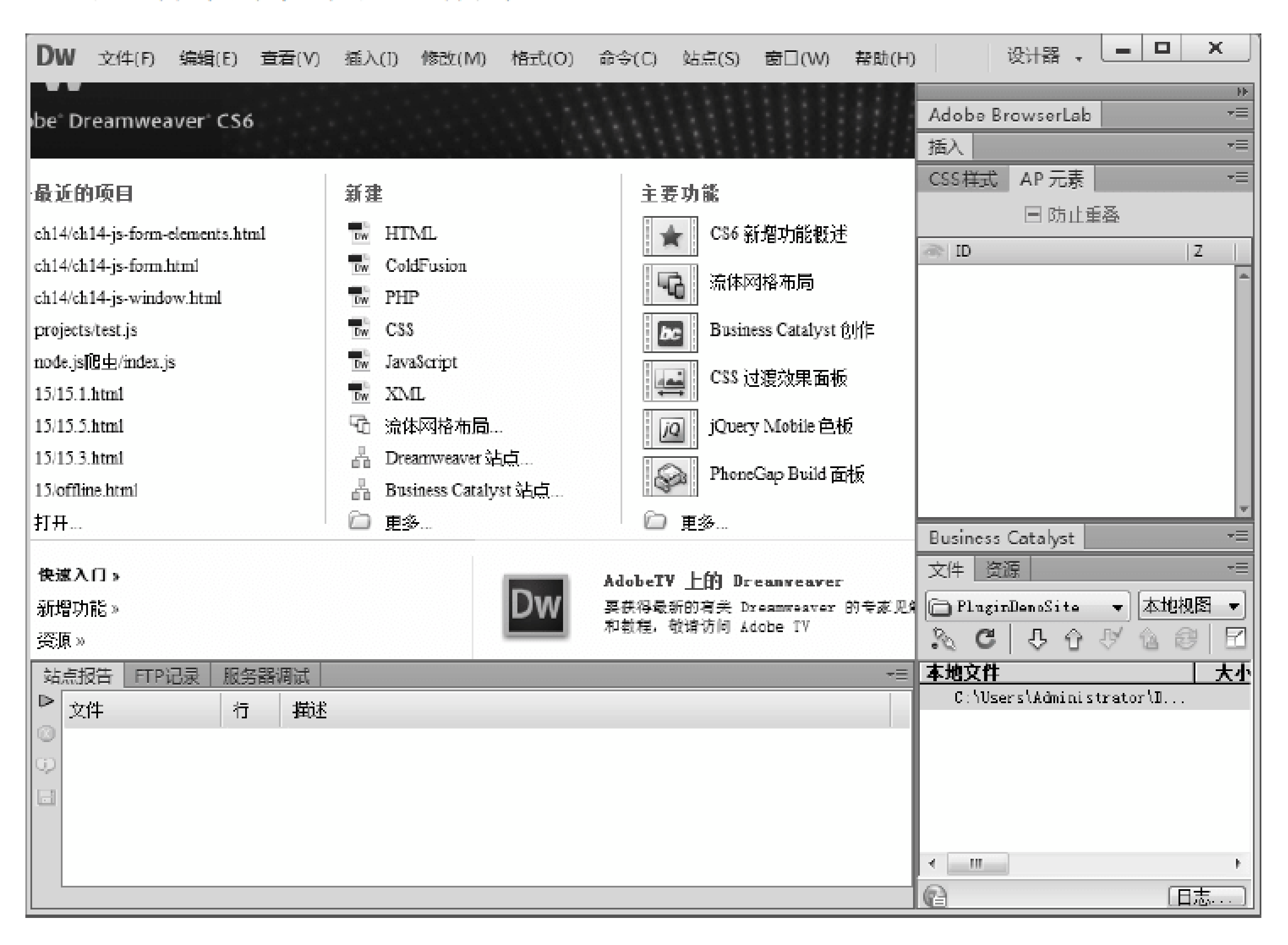


图 1.1 Adobe Dreamweaver CS6

Dreamweaver CS6 版本支持 CSS 3、HTML 5，并集成了 jQuery 代码提示功能，是网页开发人员开发大型项目或长期使用的必备工具。

2. jetBrains WebStorm

jetBrains WebStorm 是 JavaScript 集成开发平台中针对性最强、功能最完善且非常简单易学的一款开发工具。jetBrains 公司是近年来最成功的一家专业软件平台开发商，推出了多系列专业性非

常强的开发工具，在用户体验方面十分下功夫。WebStorm 开发平台就是专门针对前端开发（HTML、JavaScript、CSS）而设计的。图 1.2 就是开发平台界面的演示效果。

图 1.2　jetBrains WebStorm

3．Sublime Text

Sublime Text 是 JavaScript 开发环境 IDE 中比较漂亮（见图 1.3）的且对开发支持非常良好的一款文本编辑器，简洁、强大、高效。Sublime 做了很多用户体验方面的改进和支持，对审美有要求的读者可果断入手。

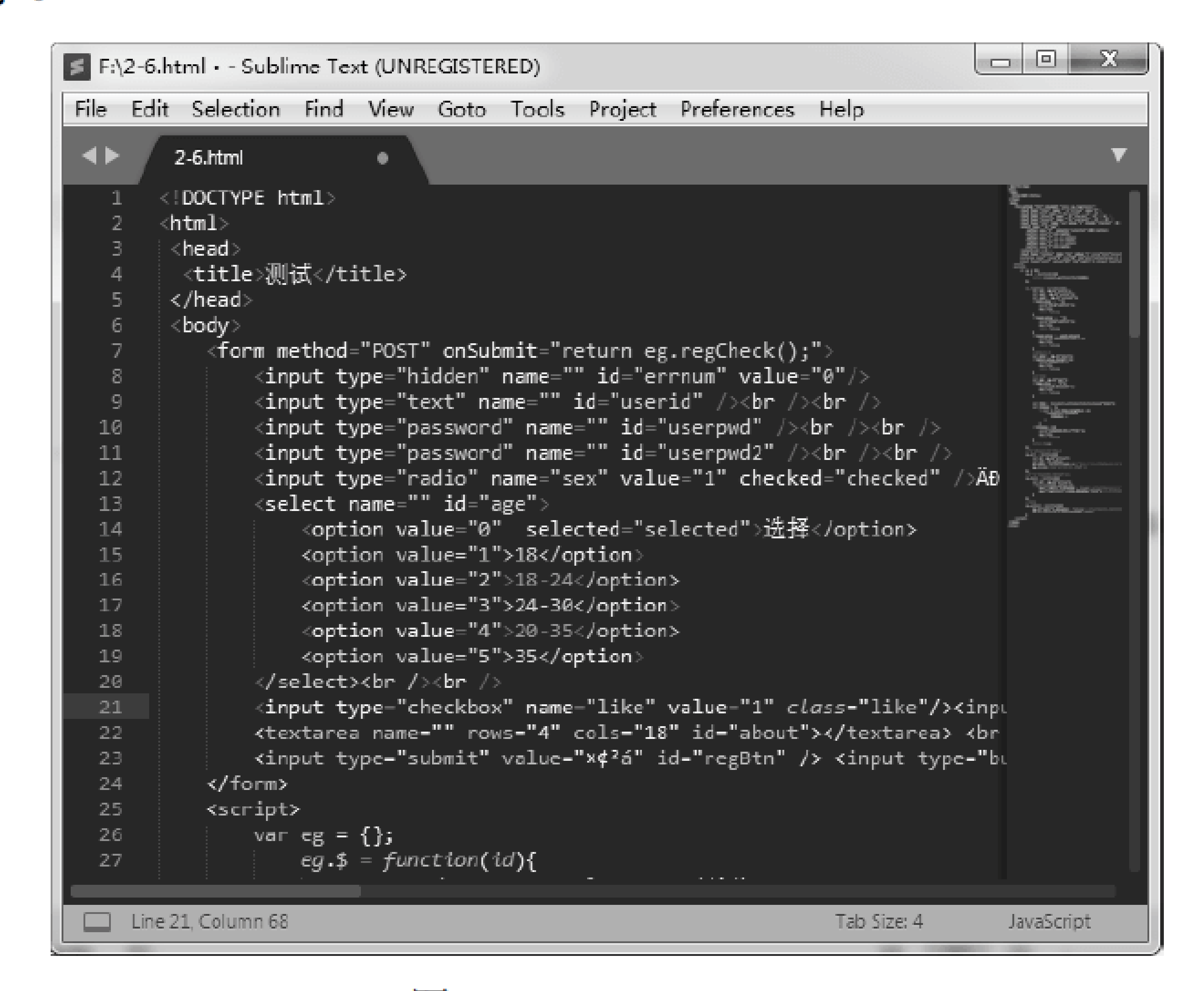


图 1.3　Sublime Text 3

1.3　JavaScript 的调试

目前的主流浏览器均支持直接调试 JavaScript 代码，本节以常见的 Chrome 和 Firefox 两款浏览器为例进行介绍。

这里先给出一段测试代码（详见源代码目录 ch01-js-debug.html 文件）：

【代码 1-1】

```
01 <!doctype html>
02 <html lang="en">
03 <head>
04   <!-- 添加文档头部内容 -->
05   <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
06   <meta http-equiv="Content-Language" content="zh-cn" />
07   <link rel="stylesheet" type="text/css" href="css/style.css">
08   <title>JavaScript 全程实例</title>
09 </head>
10 <body>
11   <!-- 添加文档主体内容 -->
12   <header>
13       <nav>JavaScript 基础 - 脚本调试</nav>
14   </header>
15   <hr>
16   <!-- 添加文档主体内容 -->
17   <div id="id-div-count">
18   </div>
19 </body>
20 <script type="text/javascript" src="js/ch01-js-debug.js"></script>
21 </html>
```

其中，【代码 1-1】中第 20 行代码引入了一个外链式 js 脚本文件，具体代码如下（详见源代码目录 ch01-js-debug.js 文件）：

【代码 1-2】

```
var v_id_div_count = document.getElementById("id-div-count");
var strLine;
for(var i=1; i<10; i++) {
    strLine = "i=" + i.toString() + "<br>";
    console.log(strLine);
    v_id_div_count.innerHTML += strLine;
}
```

下面分别使用 Chrome 和 FireFox 浏览器调试上面定义的 js 代码。

1. 使用 Chrome 调试

下面使用 Chrome 浏览器运行测试代码所在的 HTML 网页，如图 1.4 所示。打开 Chrome 浏览器的开发工具面板（或按 F12 键），如图 1.5 所示。

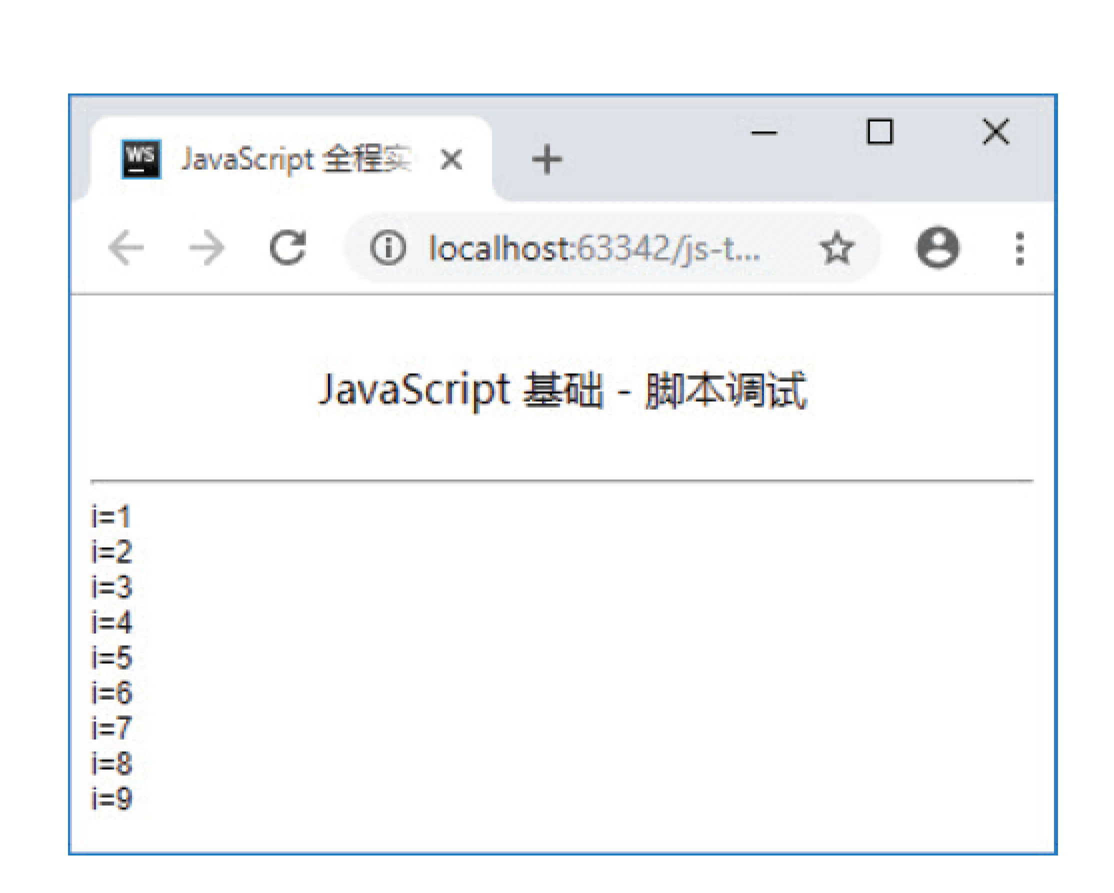


图 1.4　使用 Chrome 浏览器打开

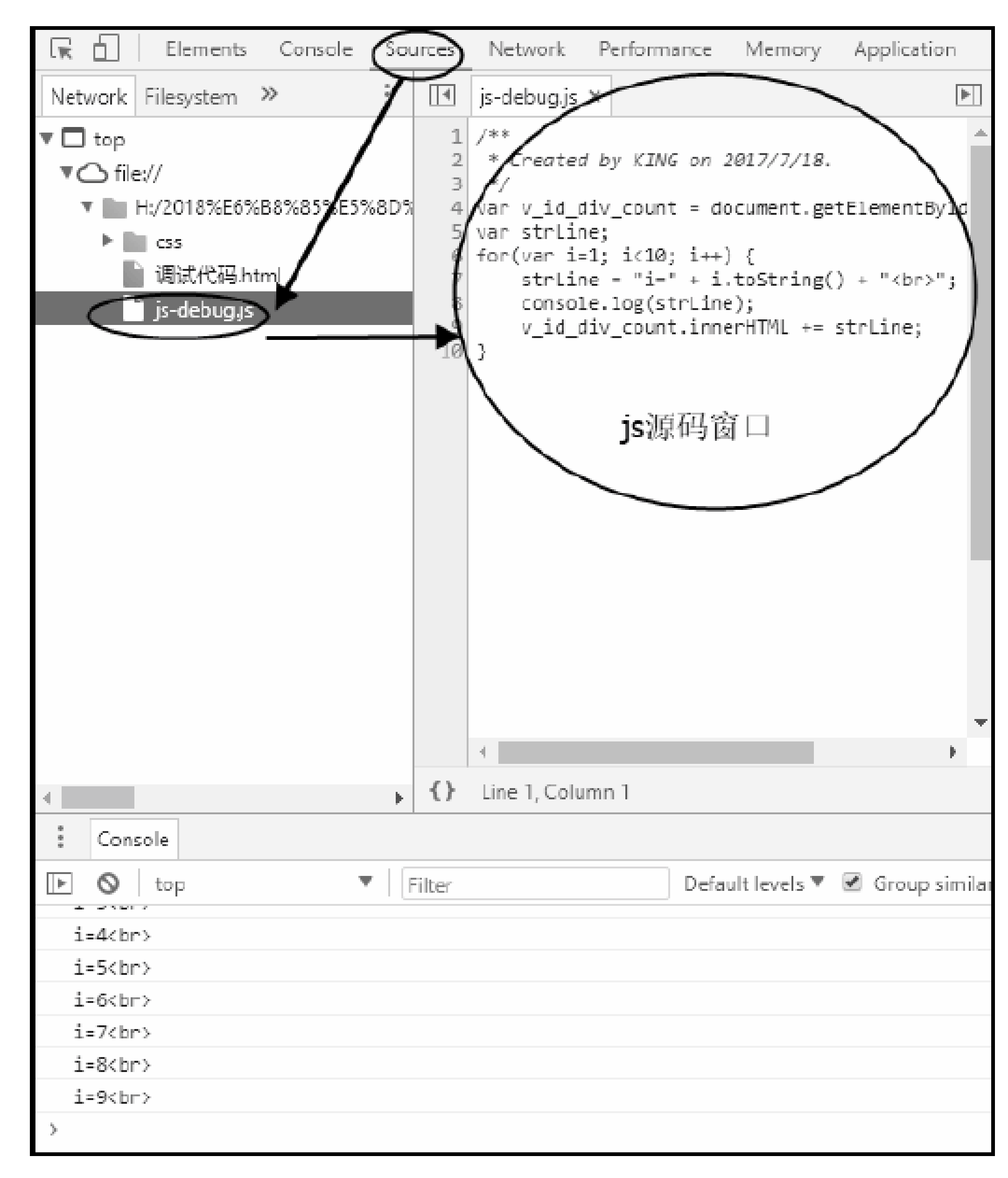


图 1.5　打开 Chrome 浏览器 js 调试功能

可以在 JavaScript 代码的第 08 行添加断点，单击行号就可以，如图 1.6 所示。这样每次程序执行到该行时中断，如图 1.7 所示。将鼠标放在某个变量上，就会显示当前变量的值，然后通过右侧的调试控制条可以控制是否继续执行下一行。



图 1.6　为 js 脚本代码设置断点

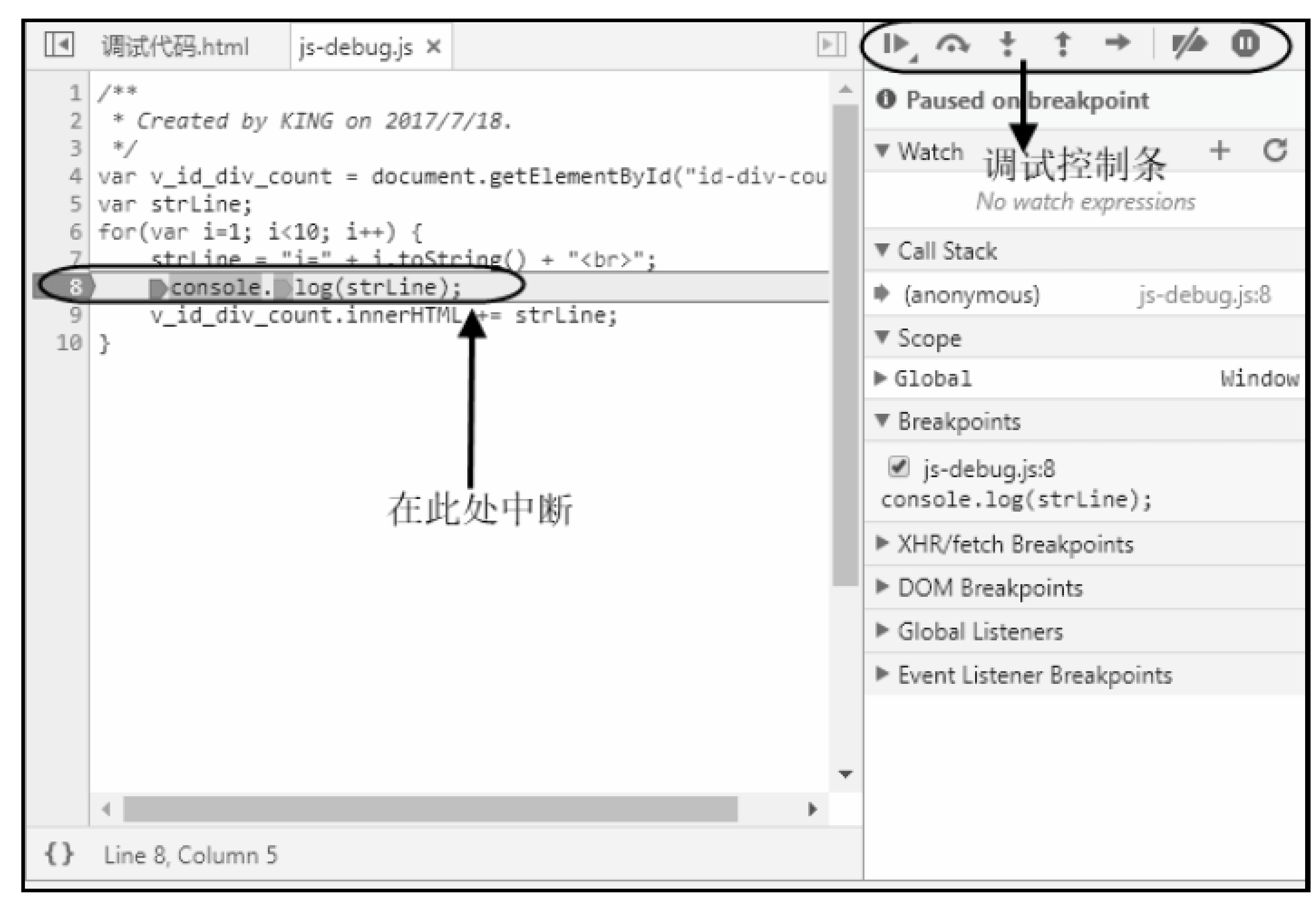


图 1.7　调试脚本代码

2．使用 Firefox 调试

下面使用 Firefox 浏览器运行测试代码所在的 HTML 网页，如图 1.8 所示。打开 Firefox 浏览器的调试功能面板，如图 1.9 所示。

下面在图 1.9 的 js 源码窗口中为【代码 1-2】中的第 05 行脚本语句设置断点，如图 1.10 所示。

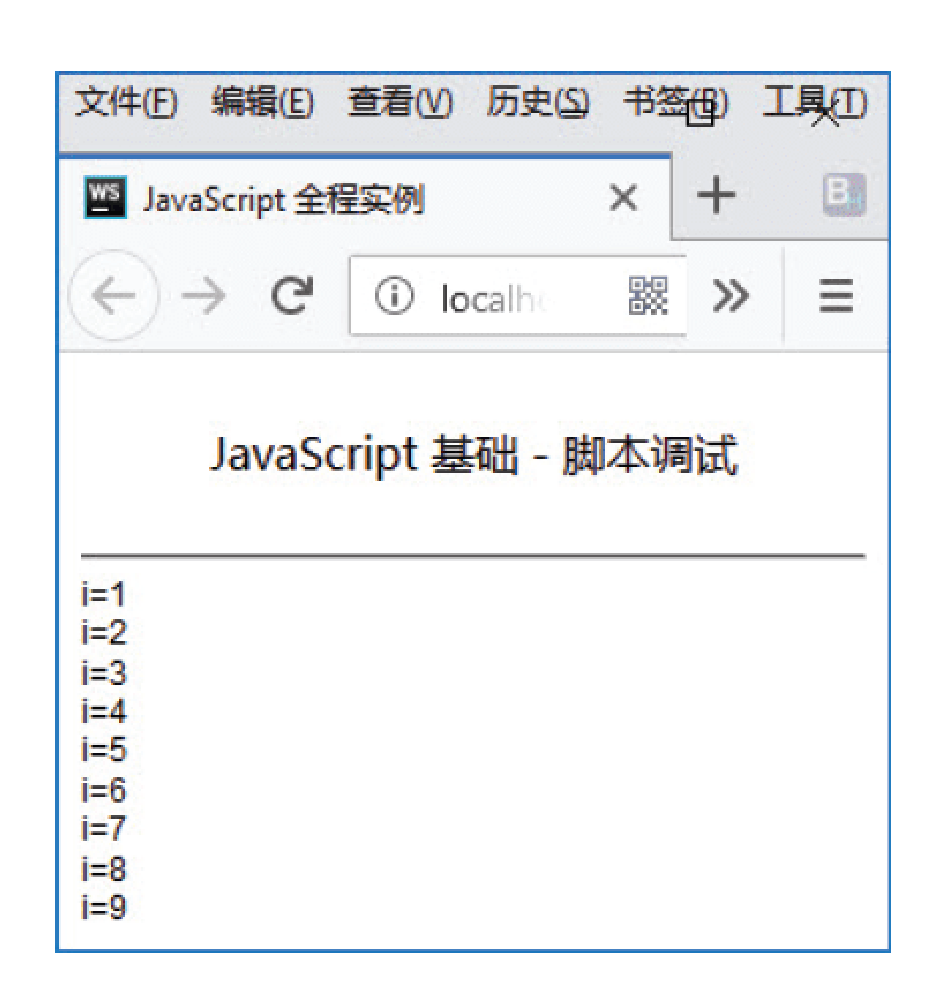


图 1.8 使用 Firefox 浏览器调试 js 脚本



图 1.9 打开 Firefox 浏览器 js 调试功能面板

然后，按“F5”功能键重新刷新页面，再按步进“F11”功能键来调试执行 js 代码，页面效果如图 1.11 和图 1.12 所示。在图 1.11 和 1.12 中可以看到，每次执行到【代码 1-2】中第 05 行脚本语句设置断点处时，js 代码均会被中断，然后在日志窗口中输出调试信息（变量“i”计数器的数值）。以上就是 JavaScript 脚本语言开发与调试的基本过程方法。

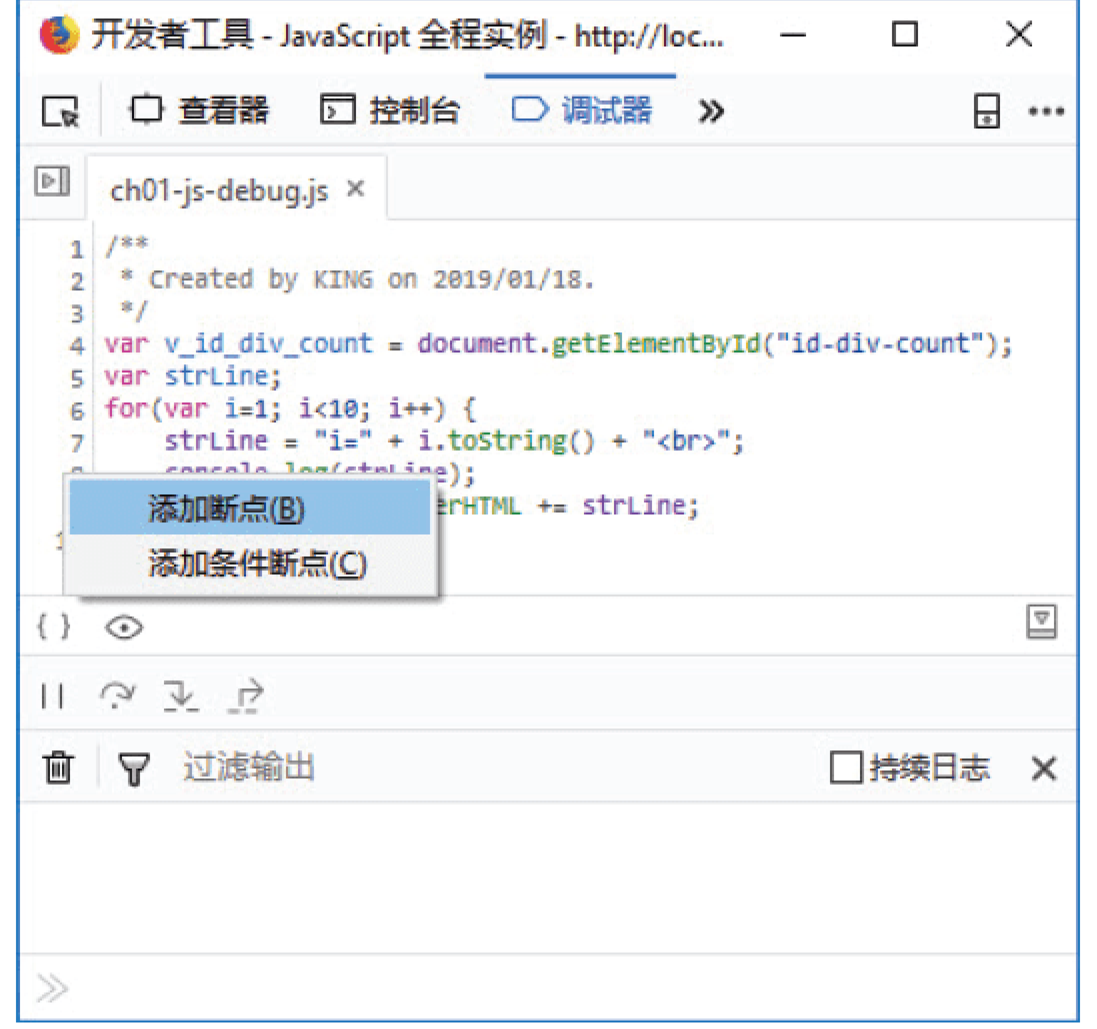



图 1.10 使用 Firefox 浏览器为 js 脚本代码设置断点

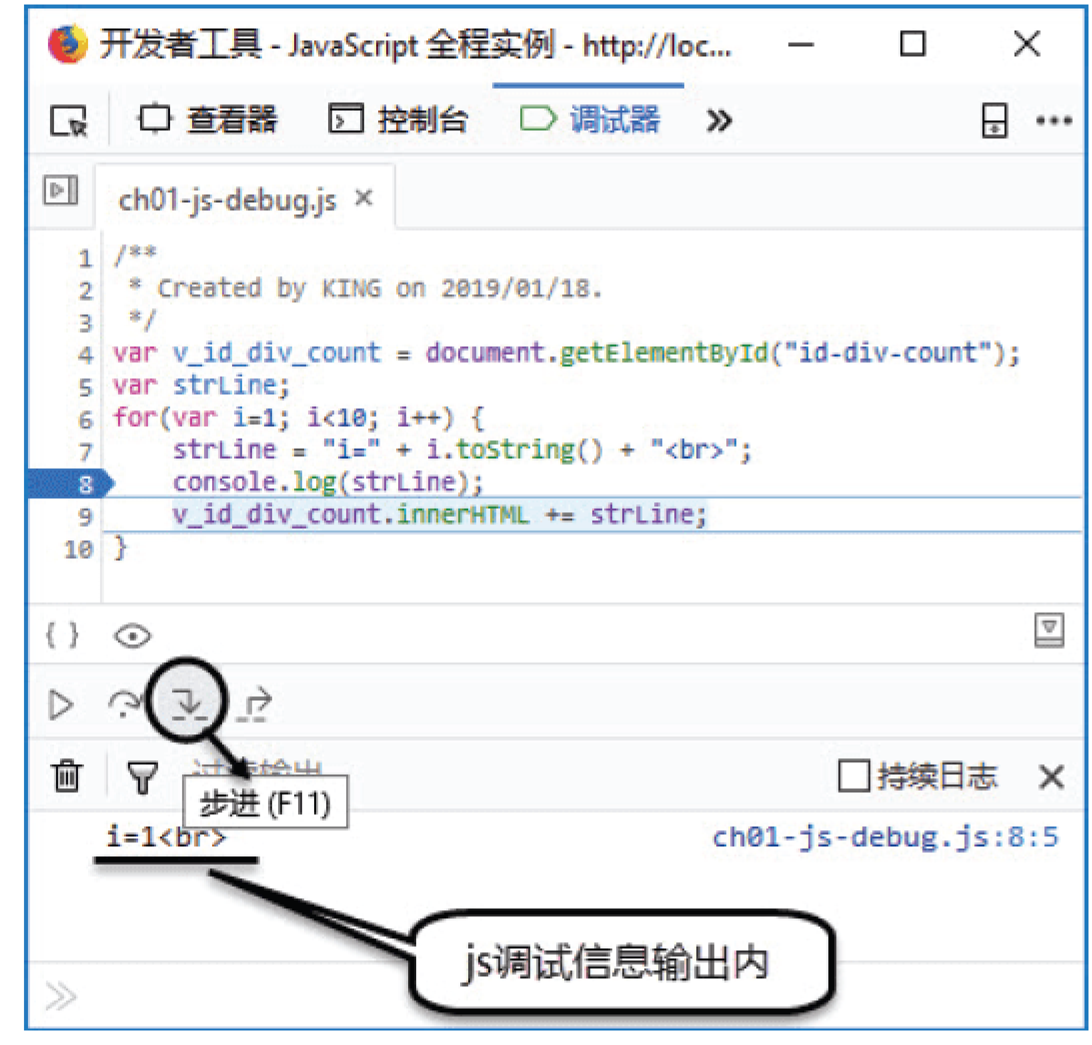


图 1.11 使用步进（F11）方式调试脚本代码

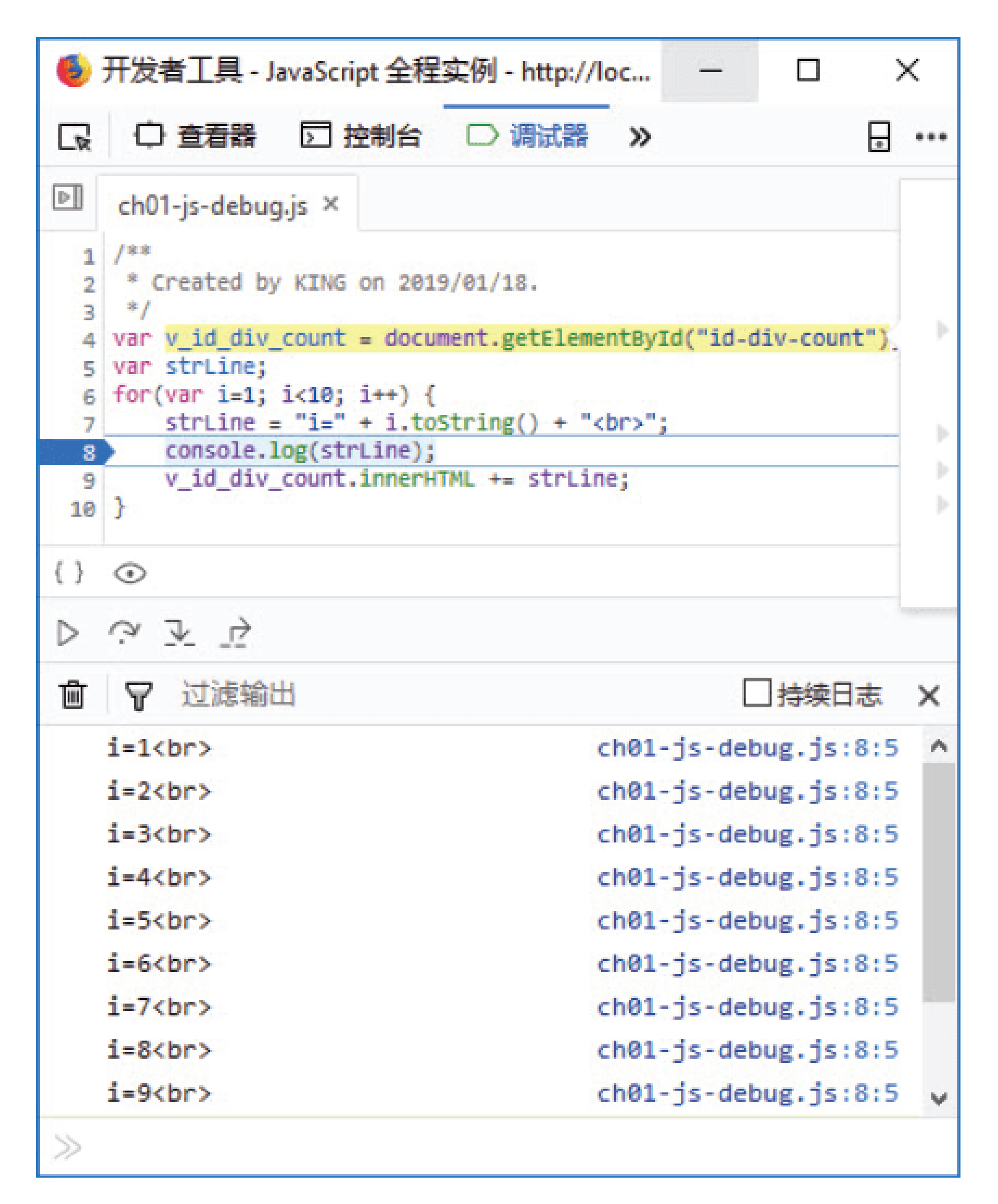


图 1.12　使用跨越（F10）方式调试脚本代码

第 2 章　JavaScript 控制表单

本章介绍如何通过 JavaScript 来控制 HTML 表单。表单是网页开发中比较常见且十分重要的一种元素，在很多应用场景下都会使用到。

2.1　JavaScript 与 HTML 表单

HTML 表单是一个包含多种表单元素（比如：文本域、下拉列表、单选框、复选框、提交按钮等）的区域。在 HTML 表单中，用户可以输入或选择多种类型的信息，然后提交到服务器来处理。

在 Web 开发中，通过 JavaScript 可以很有效地操作控制表单。例如，验证表单域中用户输入数据的合法性，格式化表单域中的用户输入数据，遍历表单域中的全部项，动态修改表单域中某项数据，控制表单的提交与重置，等等。

另外，在 HTML 5 规范中，为表单（<form>）增加了多个新的输入类型，提供了更好的输入控制和验证，与 JavaScript 结合得更加完美。

HTML 表单是通过标签（<form>）来定义的，具体语法格式如下：

【代码 2-1】

```
<form>
…form elements…
</form>
```

2.2　JavaScript 遍历表单

HTML 表单对象（Form）中定义有一个 elements 集合属性，可以返回表单中所有元素的数组集合。那么，JavaScript 通过 for 循环语句就可以实现遍历表单中全部表单项的操作了。

【代码 2-2】（详见源代码目录 ch02-js-traverse-form.html 文件）

```
<!doctype html>
<html lang="en">
<head>
  <!-- 添加文档头部内容 -->
  <title>JavaScript 全程实例</title>
```

```
06 </head>
07 <body>
08   <!-- 添加文档主体内容 -->
09   <header>
10       <nav>JavaScript 控制表单 - 遍历表单</nav>
11   </header>
12   <hr>
13   <!-- 添加文档主体内容 -->
14   <form name="formTraverse" method="get">
15       <p>First name: <input type="text" name="fname" /></p>
16       <p>Last name: <input type="text" name="lname" /></p>
17       <input type="submit" value="Submit" />
18   </form>
19 </body>
20 <script type="text/javascript">
21     var i;
22     var els = formTraverse.elements;
23     for(i in els) {
24         var el = els[i];
25         if(el.type)
26             console.log("element type : " + el.type +
                         ",element name : " + el.name);
27     }
28 </script>
29 </html>
```

关于【代码 2-2】的说明：

- 第 14～18 代码通过标签<form>定义了一个表单，内部通过标签<input>定义一组文本框和一个提交按钮。
- 第 20～28 行通过<script>标签定义了 JavaScript 脚本代码，用于实现对表单的遍历操作。
  - 第 22 行代码通过表单 form 的 elements 属性，获取了表单域中全部表单项元素的数组集合（els）。
  - 第 23～27 行代码通过 for…in…语句遍历数组集合（els），并在控制台中输出每项表单元素的 type 属性和 name 属性。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 2.1 所示。

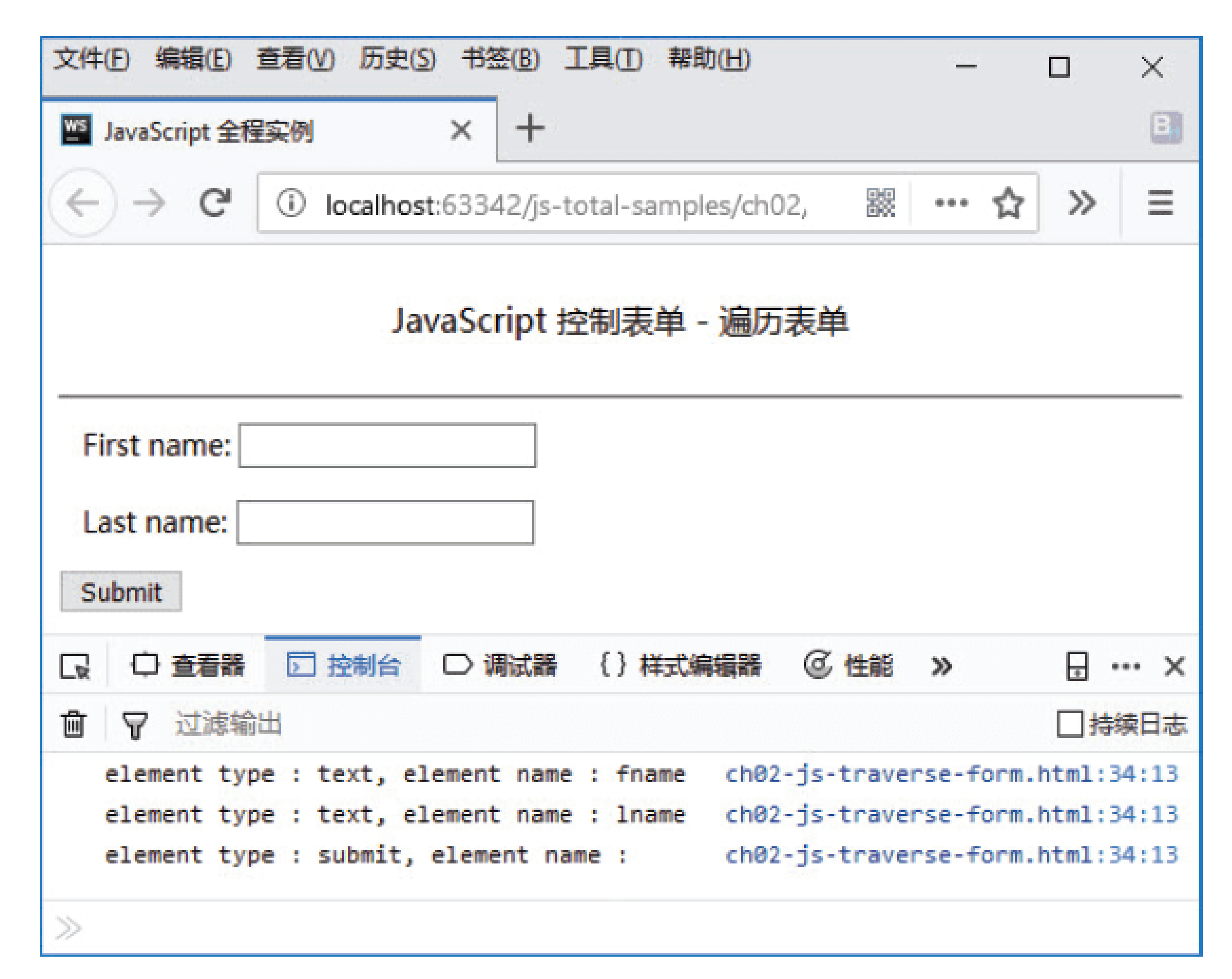


图 2.1　JavaScript 遍历表单

2.3　通过 name 和 id 访问表单元素

在 JavaScript 中，可以通过表单元素的 name 属性和 id 属性来访问该控件。对于 name 属性需要使用 getElementsByName()方法，而对于 id 属性则需要使用 getElementById()方法，这两种方法在使用上是有所区别的。

【代码 2-3】（详见源代码目录 ch02-js-get-ele.html 文件）

```
<!doctype html>
<html lang="en">
<head>
  <!-- 添加文档头部内容 -->
  <title>JavaScript 全程实例</title>
</head>
<body>
  <!-- 添加文档主体内容 -->
  <header>
      <nav>JavaScript 控制表单 - 通过 name 和 id 访问控件</nav>
  </header>
  <hr>
  <!-- 添加文档主体内容 -->
  <form name="formTraverse" method="get">
      <p>First Name: <input type="text" name="fname" id="id-fname" /></p>
      <p>Last Name: <input type="text" name="lname"  id="id-lname" /></p>
      <input type="button" id="id-get-name" onclick="on_get_name();"
             value="获取用户名" />
```

```
18    </form>
19  </body>
20  <script type="text/javascript">
21    function on_get_name() {
22         var n_fname = document.getElementsByName("fname");
23         var id_lname = document.getElementById("id-lname");
24         console.log("First Name : " + n_fname[0].value);
25         console.log("Last Name : " + id_lname.value);
26      }
27  </script>
28  </html>
```

关于【代码 2-3】的说明：

- 第 15～16 代码通过标签<input>定义一组文本框，而且同时增加了 name 属性和 id 属性的定义。
- 第 22 行代码通过 getElementsByName("fname")方法，实现了对第 15 行代码定义标签<input name="fname">的访问。不过，需要注意的是该方法返回的是一个控件对象的数组集合，所以第 24 行代码中需要先使用数组下标格式，再通过 value 属性获取用户输入的数据信息。
- 第 23 行代码是通过 getElementById("id-lname")方法，实现对第 16 行代码定义标签<input id="id-lname">的访问。该方法直接返回了该控件对象，所以第 25 行代码可以直接通过 value 属性获取用户输入的数据信息。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 2.2 所示。

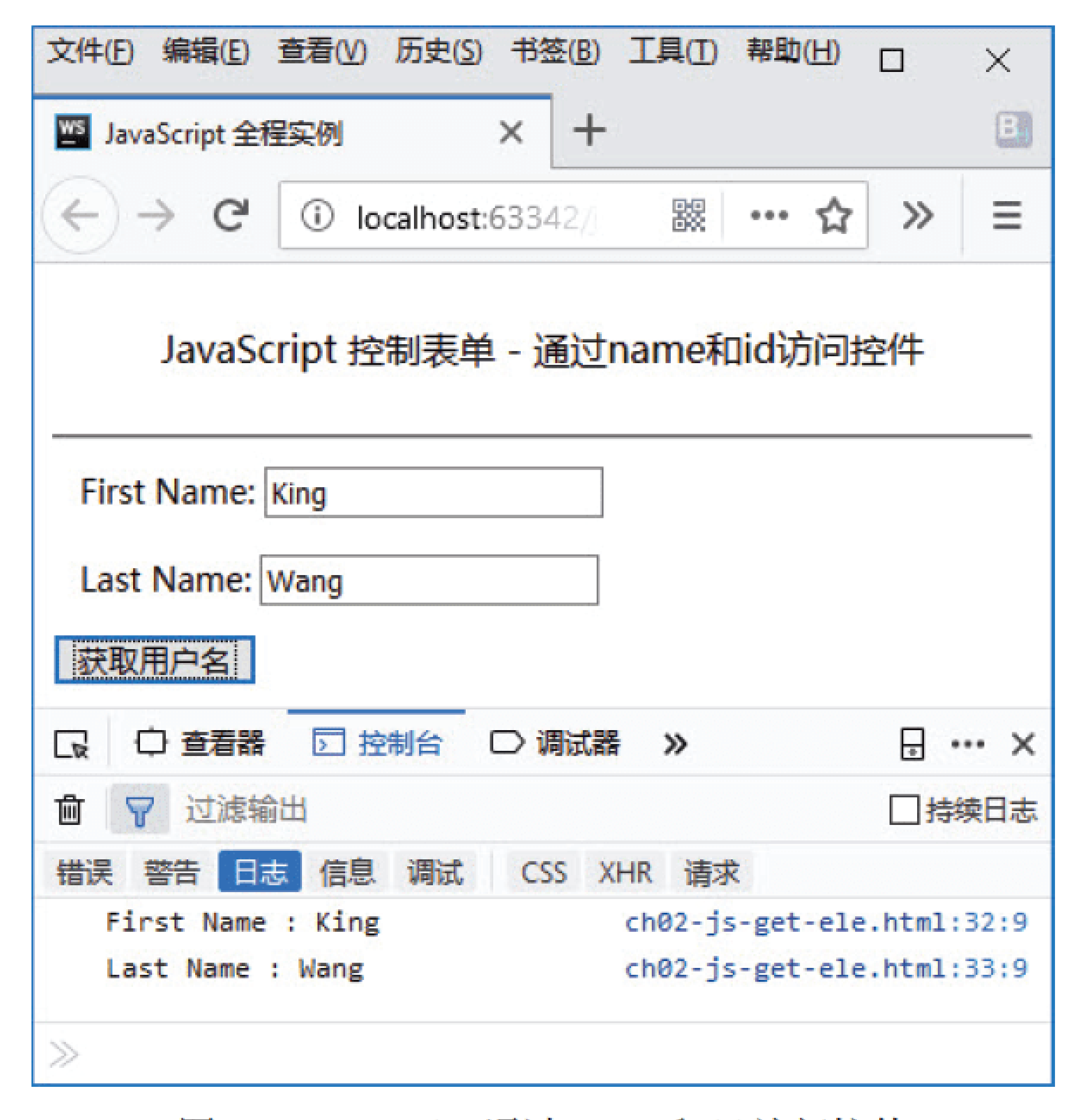


图 2.2　JavaScript 通过 name 和 id 访问控件

2.4 动态修改表单控件的值

通过 JavaScript 可以动态修改表单中控件的值，包括文本框内容、下拉菜单选项文本和按钮名称等，下面就介绍一个动态修改表单控件值的代码实例。

【代码 2-4】（详见源代码目录 ch02-js-modify-ele.html 文件）

```
<!doctype html>
<html lang="en">
<head>
  <!-- 添加文档头部内容 -->
  <title>JavaScript 全程实例</title>
</head>
<body>
  <!-- 添加文档主体内容 -->
  <header>
      <nav>JavaScript 控制表单 - 动态修改表单控件的值</nav>
  </header>
  <hr>
  <!-- 添加文档主体内容 -->
  <form name="formTraverse" method="get">
      <p>First Name: <input type="text" name="fname" id="id-fname"
              value="King" /></p>
      <p>Last Name: <input type="text" name="lname"  id="id-lname"
              value="Wang" /></p>
      <select id="id-sel-gender">
          <option value="0">请选择...</option>
          <option value="1">男</option>
          <option value="2">女</option>
      </select><br><br><br><br><br><br>
  <input type="button" id="id-modify-text" onclick="on_modify_text();"
          value="修改控件文本" />
  </form>
</body>
<script type="text/javascript">
  function on_modify_text() {
      document.getElementById("id-fname").value = "Tina";
       document.getElementById("id-lname").value = "Xi";
       var len = document.getElementById("id-sel-gender").options.length;
       for(let i=0; i<len; i++)
           document.getElementById("id-sel-gender").options[i].innerText =
```

```
32                document.getElementById("id-sel-gender").options[i].value;
33            document.getElementById("id-modify-text").value = "文本已经修改";
34        }
35    </script>
36    </html>
```

关于【代码 2-4】的说明：

- 第 14～23 代码在表单内定义了一组文本框、一个下拉菜单和一个按钮控件。
- 第 27～28 行代码通过设定新的 value 属性值，修改了第 15～16 行代码定义的文本域显示内容。
- 第 30～32 行代码通过<select>控件的 options 属性，修改了第 17～21 行代码定义的下拉菜单的显示内容。
- 第 33 行代码通过设定新的 value 属性值，修改了第 22 行代码定义的按钮名称。

下面使用 Firefox 浏览器运行测试该 HTML 网页，初始效果如图 2.3 所示。继续单击页面中的“修改控件文本”按钮，页面更新后的效果如图 2.4 所示。

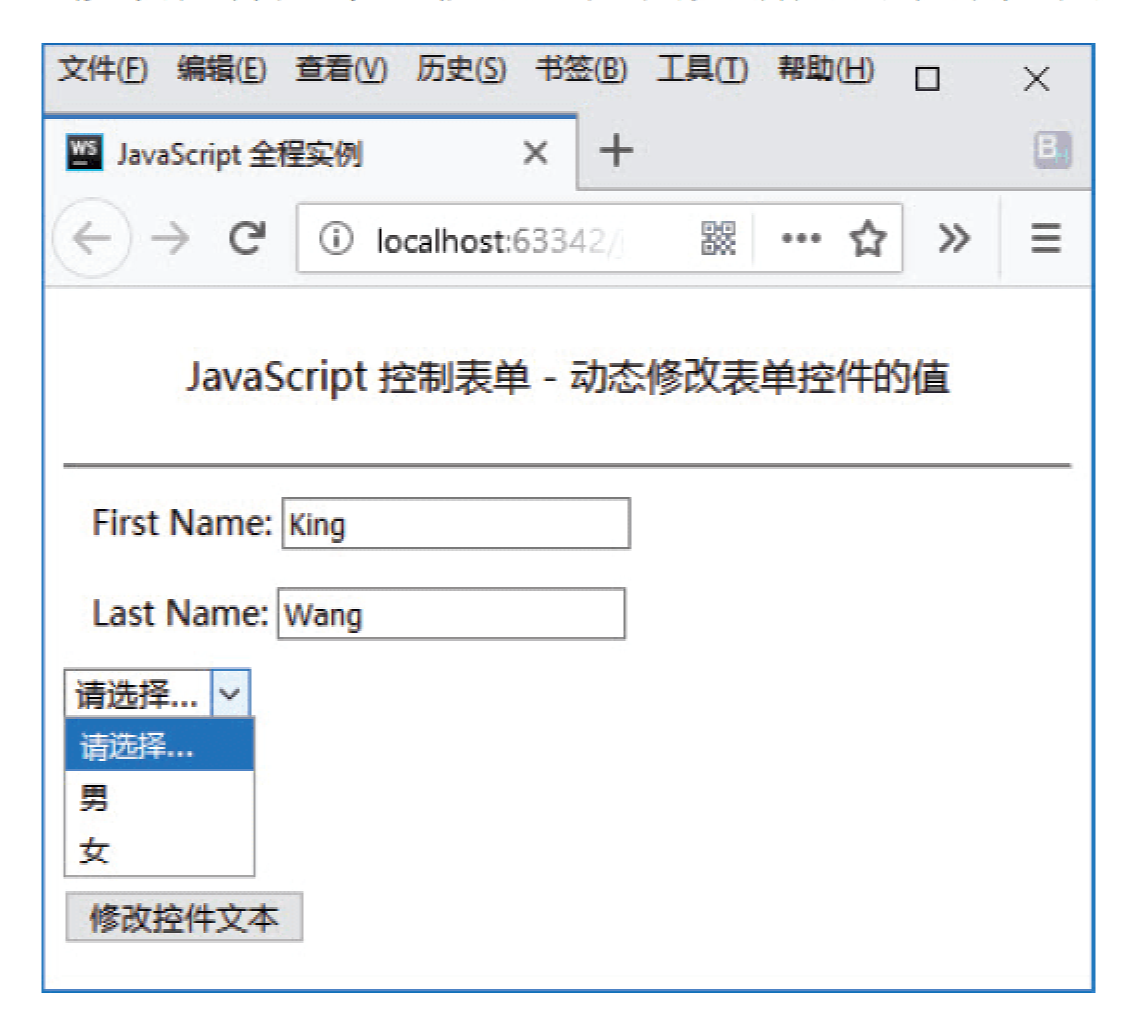


图 2.3　JavaScript 动态修改表单控件的值（一）

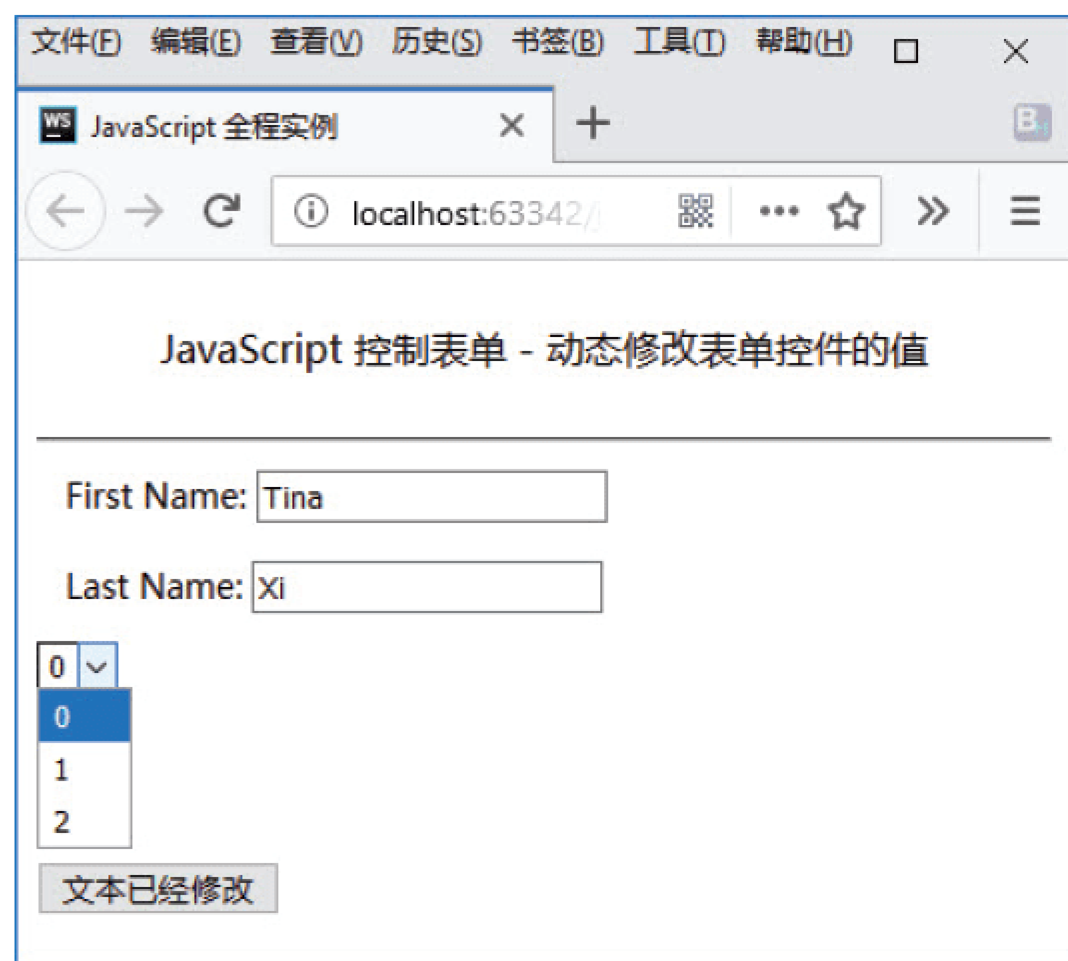


图 2.4　JavaScript 动态修改表单控件的值（二）

2.5　获取表单内文本框的数量

文本框（<input type="text">）是 HTML 表单中常用的元素，JavaScript 通过筛选累计表单中全部<input>标签的 type 属性值是否为 text 类型就可以实现获取表单中全部文本框数量的功能。

【代码 2-5】（详见源代码目录 ch02-js-input-text-count.html 文件）

```
01 <!doctype html>
02 <html lang="en">
03 <head>
04     <!-- 添加文档头部内容 -->
05     <title>JavaScript 全程实例</title>
06 </head>
07 <body>
08 <!-- 添加文档主体内容 -->
09 <header>
10     <nav>JavaScript 控制表单 - 获取表单内文本框的数量</nav>
11 </header>
12 <hr>
13 <!-- 添加文档主体内容 -->
14 <form name="" method="get">
15     <p>First name: <input type="text" name="fname"/></p>
16     <p>Middle name: <input type="text" name="mname"/></p>
17     <p>Last name: <input type="text" name="lname"/></p>
18     <input type="submit" value="Submit" />
19     <input type="reset" value="Reset" />
20 </form>
21 </body>
22 <script type="text/javascript">
23     var i, iCounts = 0;
24     var inputs = document.getElementsByTagName("input");
25     for (i = 0; i < inputs.length; i++) {
26         var el = inputs[i];
27         if (el.type == "text")
28             iCounts++;
29     }
30     console.log("表单内文本框的数量: " + iCounts);
31 </script>
32 </html>
```

关于【代码 2-5】的说明：

- 第 14～20 代码在表单内定义了一组文本框（共计 3 个）和一组按钮控件（共计 2 个）。
- 第 27～28 行代码通过 for 循环语句，依次判断<input>标签控件的 type 属性值是否为“text”来甄别该控件是否为文本框。如果判断结果为“true”，就通过累加器变量 iCounts 计数。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 2.5 所示。



图 2.5 JavaScript 获取表单内文本框的数量

2.6 修改表单的提交方式

HTML 表单<form>中定义有一个 method 属性，用于规定提交表单数据的方式(GET 和 POST)。JavaScript 可以通过修改 method 属性值改变表单的提交方式，下面给出一个修改表单提交方式的代码实例。

【代码 2-6】(详见源代码目录 ch02-js-modify-form-method.html 文件)

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 控制表单 - 修改表单的提交方法</nav>
</header>
<hr>
<!-- 添加文档主体内容 -->
<form name="formMethod" method="" action="#">
    <p>请选择表单提交方法:
        <select name="selMethod">
            <option value="">请选择...</option>
```

```
18             <option value="get">get 方式</option>
19             <option value="post">post 方式</option>
20         </select>
21     </p>
22     <input type="button" value="修改提交方法" onclick="modifyMethod()"/>
23 </form>
24 </body>
25 <script type="text/javascript">
26     function modifyMethod() {
27         var fMethod = document.formMethod;         // TODO: 获取表单 DOM
28         var selMethod = fMethod.selMethod.value;   // TODO: 选择的方法
29         fMethod.method = selMethod;
30         console.log("表单当前选择的提交方法: " + fMethod.method);
31     }
32 </script>
33 </html>
```

关于【代码 2-6】的说明：

- 第 16～20 代码在表单内定义了下拉选择框（selMethod），定义了“get”和“post”两种表单提交方式的选择项。
- 第 28～29 行代码先获取了下拉选择框（selMethod）的选项值，然后通过该值修改（重新定义）表单（fMethod）的“method”属性，从而实现了对表单提交方法修改的功能。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 2.6 和图 2.7 所示。



图 2.6　JavaScript 修改表单的提交方法（get）



图 2.7　JavaScript 修改表单的提交方法（post）

2.7　动态指定表单的提交方式

在前一个代码实例的基础上，通过 JavaScript 可以进一步动态指定表单的提交方式，具体就是通过操作表单<form>对象的 submit()方法来实现的。

【代码 2-7】（详见源代码目录 ch02-js-form-dyn-method.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 控制表单 - 动态指定表单的提交方式</nav>
</header>
<hr>
<!-- 添加文档主体内容 -->
<form name="formMethod" method="" action="ch02-js-form-dyn-submit.php">
    <p>First name: <input type="text" name="fname"/></p>
    <p>Last name: <input type="text" name="lname"/></p>
    <p>请选择表单提交方法:
        <select name="selMethod">
            <option value="">请选择...</option>
            <option value="get">get 方式</option>
            <option value="post">post 方式</option>
        </select>
    </p>
    <input type="button" value="修改提交方法" onclick="modifyMethod()"/>
</form>
</body>
<script type="text/javascript">
    function modifyMethod() {
        var fMethod = document.formMethod;         // TODO: 获取表单 DOM
        var selMethod = fMethod.selMethod.value;      // TODO: 选择的方法
        console.log("表单当前选择的提交方法: " + selMethod);
        fMethod.method = selMethod;     // TODO: 修改选择的提交方法
        fMethod.submit();
    }
```

```
35 </script>
36 </html>
```

【代码 2-7】在【代码 2-6】的基础上做了如下修改：

- 第 14 行代码在表单（fMethod）的声明定义中通过“action”属性指定提交的服务端 PHP 文件（"ch02-js-form-dyn-submit.php"）。
- 第 15～26 代码增加定义了一组文本框，用于输入用户的名字。
- 第 33 行代码通过使用表单（fMethod）的“submit()”方法，从而实现了动态指定表单提交方式并进行提交的功能。

下面使用 Firefox 浏览器运行测试该 HTML 网页，使用“get”方式提交表单的效果如图 2.8 所示。

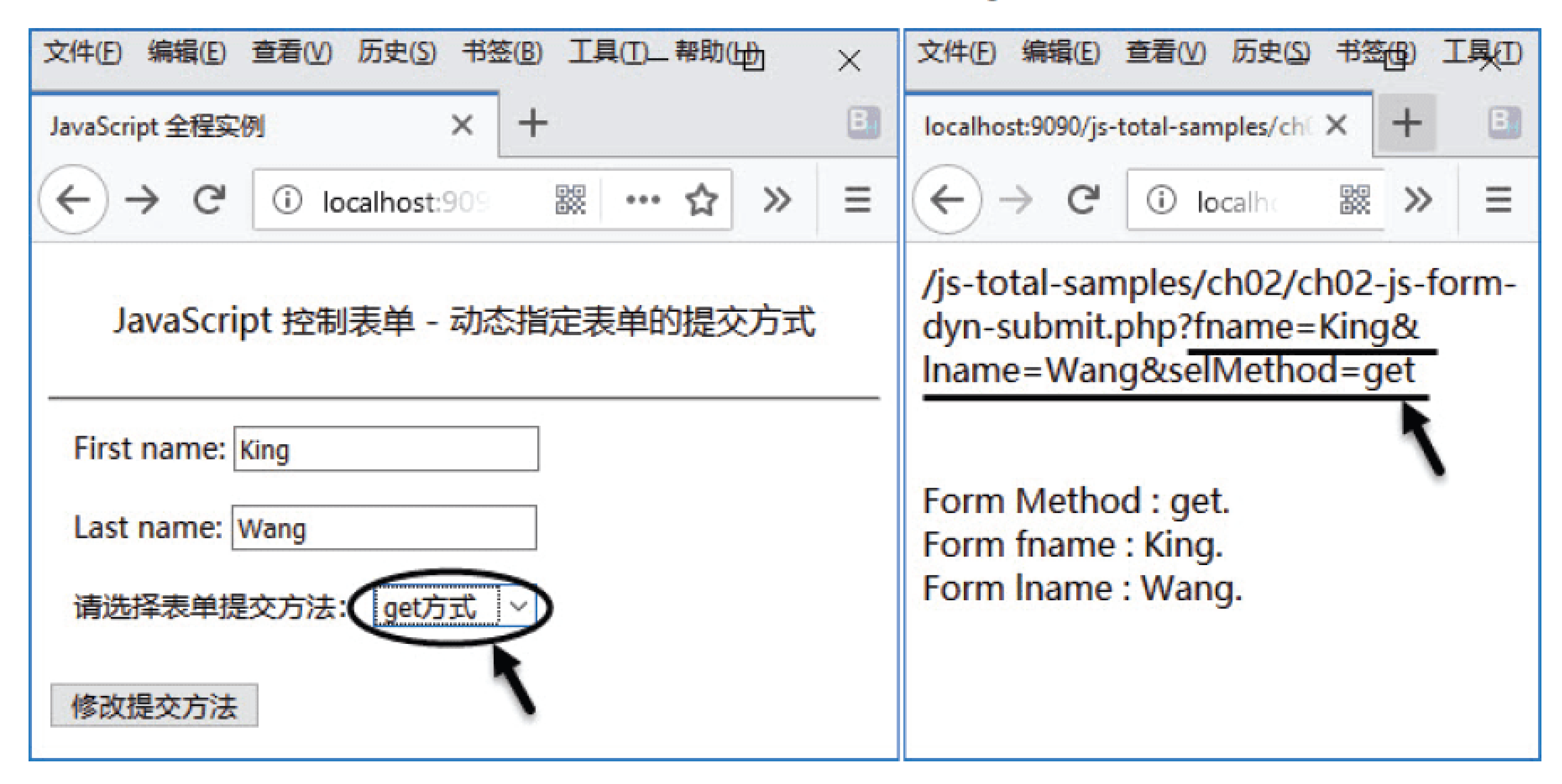


图 2.8　JavaScript 动态指定表单的提交方式（get）

如图 2.8 所示，使用“get”方式提交表单时，url 地址中是显示“query string”参数信息的。然后使用“post”方式重新提交一下表单，效果如图 2.9 所示。

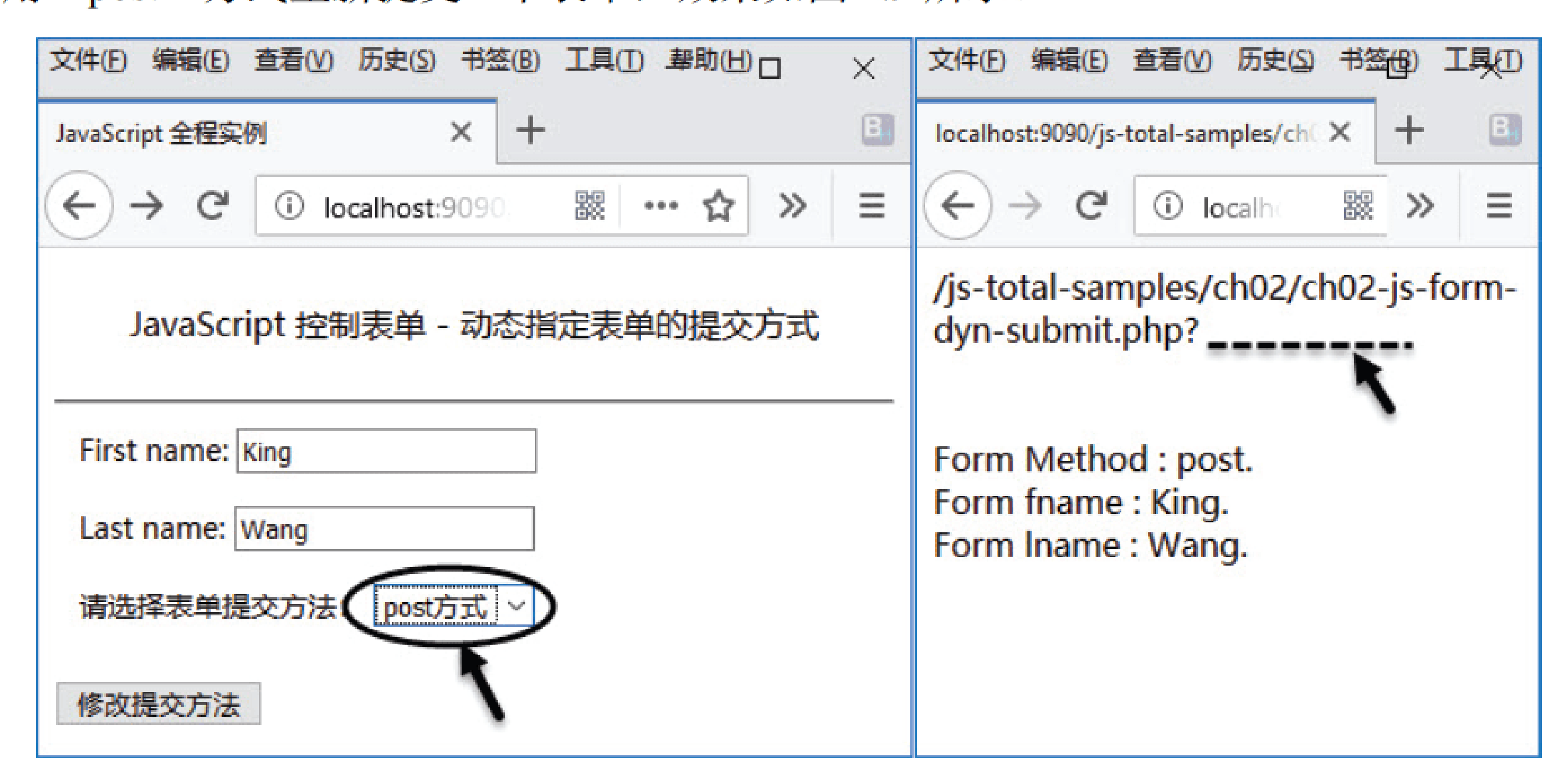


图 2.9　JavaScript 动态指定表单的提交方式（post）

如图 2.9 所示，使用“post”方式提交表单时，url 地址中是隐藏“query string”参数信息的。

2.8　动态设置焦点控件

HTML DOM 对象中定义有一个 focus()方法，用于为控件设置焦点，JavaScript 可以通过该方法实现动态设置控件焦点的操作。

【代码 2-8】（详见源代码目录 ch02-js-set-focus-ele.html 文件）

```
01 <!doctype html>
02 <html lang="en">
03 <head>
04     <!-- 添加文档头部内容 -->
05     <title>JavaScript 全程实例</title>
06 </head>
07 <body onload="on_load_set_focus()">
08 <!-- 添加文档主体内容 -->
09 <header>
10     <nav>JavaScript 控制表单 - 动态设置焦点控件</nav>
11 </header>
12 <hr>
13 <!-- 添加文档主体内容 -->
14 <form name="formTraverse" method="get">
15     <p>First Name: <input type="text" name="fname" id="id-fname"
                  value="King"/></p>
16     <p>Last Name: <input type="text" name="lname" id="id-lname"
                  value="Wang"/></p>
17     <input type="button" id="id-get-focus-ele-fname"
         onclick="on_load_set_focus('id-fname');" value="动态设置 fname 焦点"/>
18     <input type="button" id="id-get-focus-ele-lname"
         onclick="on_load_set_focus('id-lname');" value="动态设置 lname 聚焦"/>
19 </form>
20 </body>
21 <script type="text/javascript">
22     function on_load_set_focus(thisid) {
23         document.getElementById(thisid).focus();
24     }
25 </script>
26 </html>
```

关于【代码 2-8】的说明：

- 第 15 ~ 16 行代码在表单内定义了一组文本框（共计 2 个），用于测试动态设置焦点的控件。
- 第 17 ~ 18 行代码定义了一组按钮控件（共计 2 个），用于实现对文本框动态设置焦点的操作。

- 第 22 ~ 24 行代码是“on_load_set_focus()”方法的实现，通过对获取的元素对象使用“focus()”方法来实现对该元素动态设置焦点的功能。

2.9 动态获取焦点控件

前一个实例中实现了动态设置焦点控件的操作，下面继续实现动态获取焦点控件的操作。在 HTML DOM 对象中定义有一个只读的“document.activeElement”属性，可以返回 HTML 文档中当前取得焦点的元素。下面介绍一个通过 JavaScript 动态获取焦点控件的代码实例。

【代码 2-9】（详见源代码目录 ch02-js-get-focus-ele.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body onclick="on_active_click();">
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 控制表单 - 动态获取焦点控件</nav>
</header>
<hr>
<!-- 添加文档主体内容 -->
<form name="formTraverse" method="get">
    <p>First Name: <input type="text" name="fname" id="id-fname"
              value="King" /></p>
    <p>Last Name: <input type="text" name="lname" id="id-lname"
              value="Wang" /></p>
    <input type="button" id="id-get-focus-ele" value="动态获取焦点控件"/>
</form>
</body>
<script type="text/javascript">
    function on_active_click() {
        console.log(document.activeElement.id);
    }
</script>
</html>
```

关于【代码 2-9】的说明：

- 第 07 行代码为<body>元素定义了单击“onclick”事件方法（on_active_click()），这样当单击表单中的控件时会触发该“onclick”事件。
- 第 14 ~ 18 代码在表单内定义了一组文本框（共计 2 个）和一个按钮，用于测试动态获取焦点的控件。
- 第 21 ~ 23 行代码是“on_active_click()”方法的实现，第 22 行代码通过“document.activeElement.id”属性动态获取焦点控件的 id 属性。

下面使用 Firefox 浏览器运行测试该 HTML 网页，依次通过点击操作使得表单中的控件获取焦点，具体效果如图 2.10 所示。

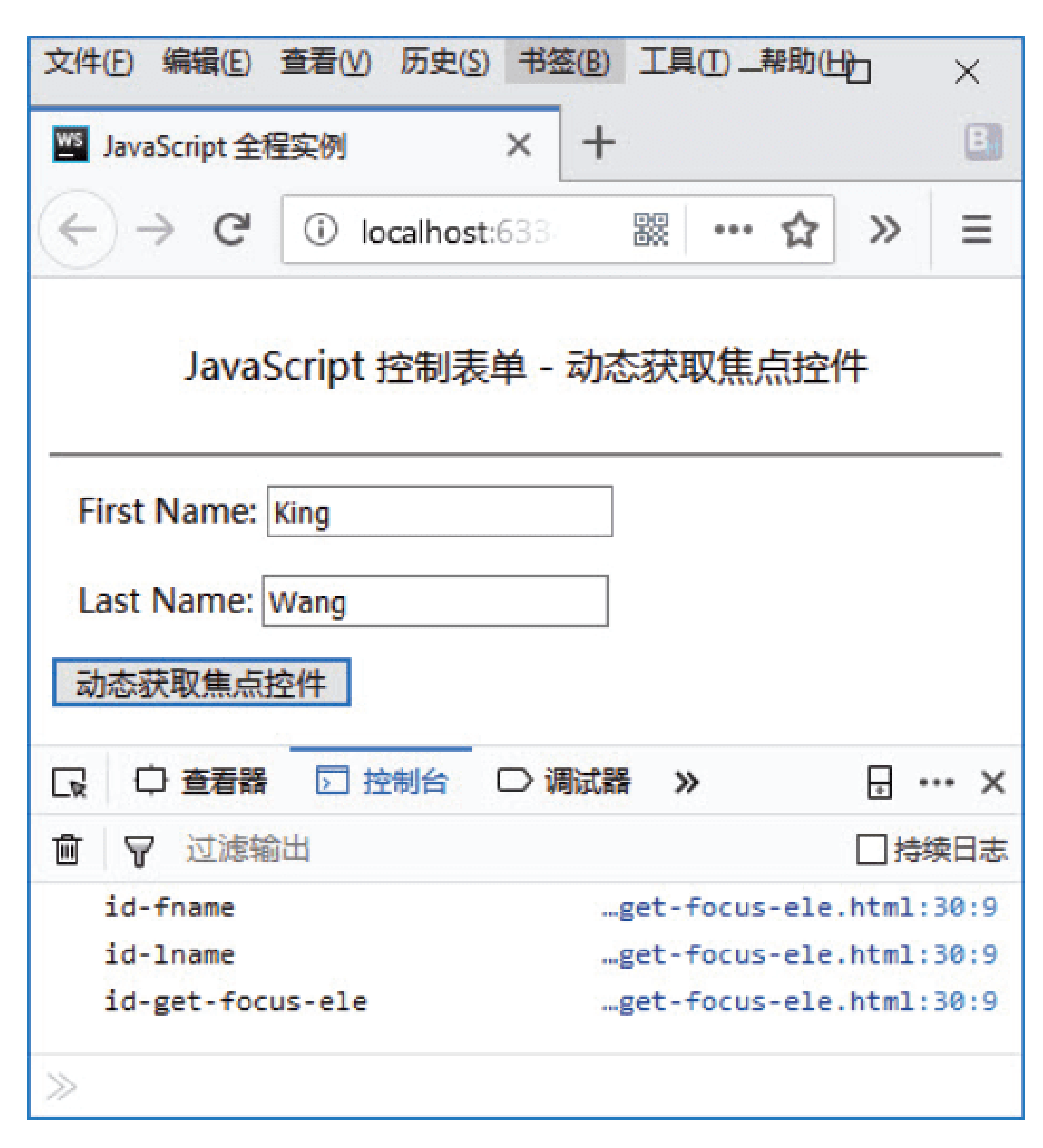


图 2.10　JavaScript 动态获取焦点控件

2.10　初始化表单里的所有控件

在某些设计场景下，HTML 页面加载后需要为表单中的控件初始化默认值。要实现该功能，既可以通过 HTML 表单控件的属性来实现，也可以通过 JavaScript 来动态操作。下面介绍一个通过 JavaScript 实现初始化表单里的所有控件的代码实例。

【代码 2-10】（详见源代码目录 ch02-js-init-form-ele.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
```

```
    <title>JavaScript 全程实例</title>
</head>
<body onload="initAllEle();">
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 控制表单 - 初始化表单里的所有控件</nav>
</header>
<hr>
<!-- 添加文档主体内容 -->
<form name="formTraverse" method="get">
    <p>First Name: <input type="text" name="fname" id="id-fname"/></p>
    <p>Last Name: <input type="text" name="lname" id="id-lname"/></p>
    <p>请选择性别:
        <select name="selMethod">
            <option value="">请选择...</option>
            <option value="get">男性</option>
            <option value="post">女性</option>
        </select>
    </p>
    <p>Birth: <input type="date" name="birth" id="id-birth"/></p>
    <p>Email: <input type="email" name="email" id="id-email"/></p>
    <p>Hobby: <br>
        <input type="checkbox" name="hobby" value="sport"/>Sport<br>
        <input type="checkbox" name="hobby" value="reading"/>Reading<br>
        <input type="checkbox" name="hobby" value="tour"/>Tour<br>
        <input type="checkbox" name="hobby" value="music"/>Music<br>
        <input type="checkbox" name="hobby" value="movie"/>Movie<br>
        <input type="checkbox" name="hobby" value="dance"/>Dance<br>
    </p>
    <input type="submit" value="提交"/>
</form>
</body>
<script type="text/javascript">
    function initAllEle() {
        // TODO: text
        document.getElementById("id-fname").value = "King";
        document.getElementsByName("lname")[0].value = "Wang";
        // TODO: select
        document.getElementsByName("selMethod")[0].selectedIndex = 1;
        // TODO: 创建一个当前日期对象
        var now = new Date();
        // TODO: 格式化日，如果小于 9，前面补 0
        var day = ("0" + now.getDate()).slice(-2);
```

```
48          // TODO: 格式化月，如果小于 9，前面补 0
49          var month = ("0" + (now.getMonth() + 1)).slice(-2);
50          // TODO: 完整日期格式
51          var today = now.getFullYear() + "-" + (month) + "-" + (day);
52          document.getElementById("id-birth").value = today;
53          // TODO: email
54          document.getElementById("id-email").value = "king@email.com";
55          // TODO: checkbox
56          var checkHobby = document.getElementsByName("hobby");
57          for (var i in checkHobby)
58              checkHobby[i].checked = true;
59      }
60  </script>
61  </html>
```

关于【代码 2-10】的说明：

- 第 07 行代码为<body>元素定义了页面加载“onload”事件方法（initAllEle()），用于在页面加载完成后对页面中的控件进行初始化操作。
- 第 14～35 行代码在表单内定义了若干控件，包括文本框、下拉选择框、Email 文本框、日期文本框和复选框，用于测试控件初始化操作。
- 第 38～59 行代码是“initAllEle()”方法的实现，通过 JavaScript 进行文本框、下拉选择框、Email 文本框、日期文本框和复选框的初始化操作。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 2.11 所示。

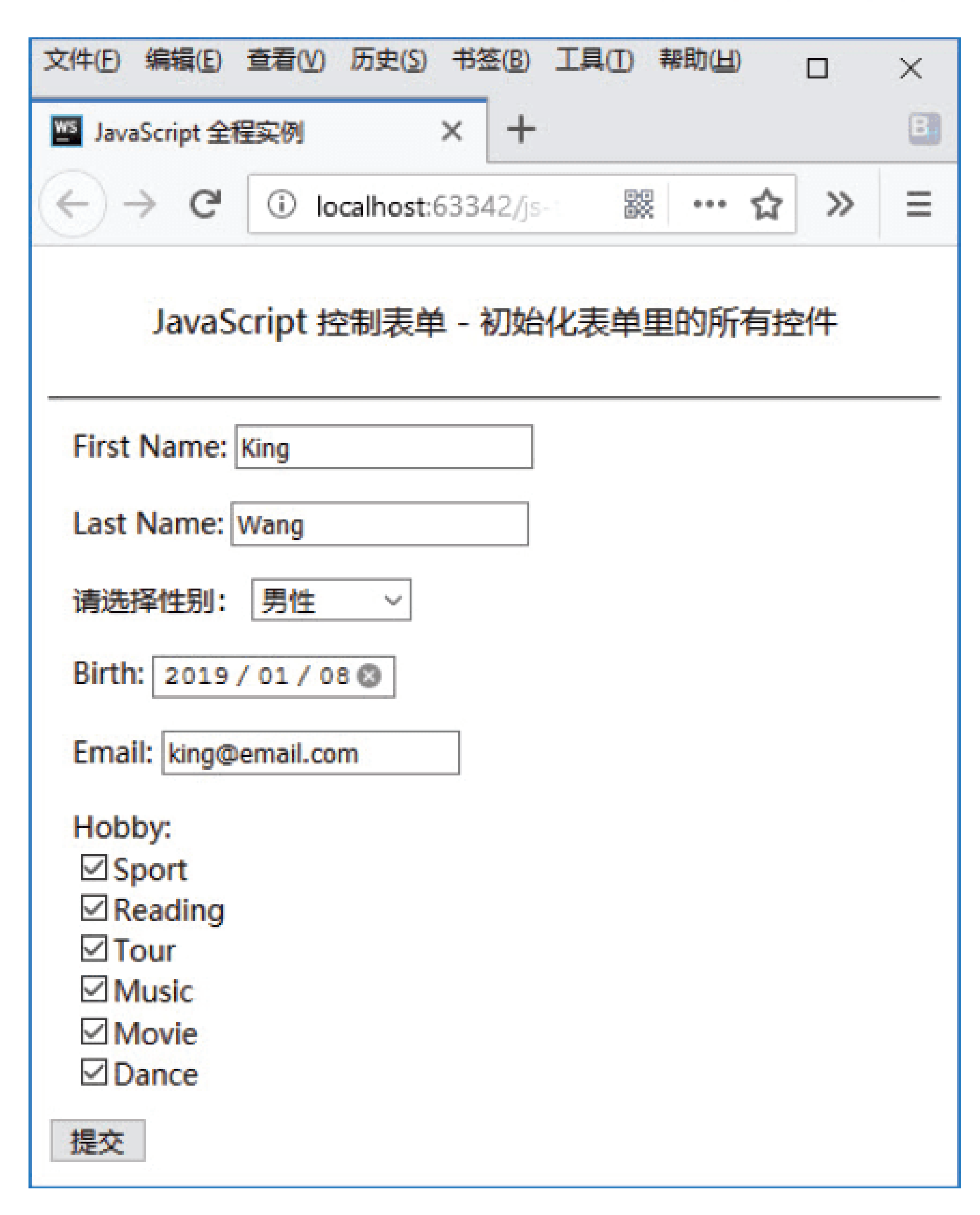


图 2.11　JavaScript 初始化表单里的所有控件

2.11 复选框全选、取消及判断是否选中的方法

其实在前面表单控件初始化的实例中已经使用了复选框全部选中的操作了。下面就完整地讲解一个如何通过 JavaScript 脚本实现复选框的全选、取消、部分选中及判断是否选中方法的代码实例。

【代码 2-11】（详见源代码目录 ch02-js-checkbox-oper.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 控制表单 - 复选框全选、取消及判断是否选中的方法</nav>
</header>
<hr>
<!-- 添加文档主体内容 -->
<form name="formTraverse" method="get">
    <p>Hobby (复选框 checkbox): <br>
        <input type="checkbox" name="hobby" value="sport"/>Sport<br>
        <input type="checkbox" name="hobby" value="reading"/>Reading<br>
        <input type="checkbox" name="hobby" value="tour"/>Tour<br>
        <input type="checkbox" name="hobby" value="music"/>Music<br>
        <input type="checkbox" name="hobby" value="movie"/>Movie<br>
        <input type="checkbox" name="hobby" value="dance"/>Dance<br>
    </p>
    <input type="button" onclick="on_checkbox_all_click()"
            value="全部选择"/>
    <input type="button" onclick="on_checkbox_none_click()"
            value="全部取消"/>
    <input type="button" onclick="on_checkbox_sel_click()"
            value="部分选择"/>
    <input type="button" onclick="on_checkbox_checked_click()"
            value="判断是否被选中"/>
</form>
</body>
<script type="text/javascript">
```

```
30      function on_checkbox_all_click() {
31          // TODO: checkbox
32          var checkHobby = document.getElementsByName("hobby");
33          for (var i in checkHobby)
34              checkHobby[i].checked = true;
35      }
36      function on_checkbox_none_click() {
37          // TODO: checkbox
38          var checkHobby = document.getElementsByName("hobby");
39          for (var i in checkHobby)
40              checkHobby[i].checked = false;
41      }
42      function on_checkbox_sel_click() {
43          // TODO: checkbox
44          var bChecked = [true, false, true, true, false, true];
45          var checkHobby = document.getElementsByName("hobby");
46          for (var i in checkHobby)
47              checkHobby[i].checked = bChecked[i];
48      }
49      function on_checkbox_checked_click() {
50          // TODO: checkbox
51          var checkHobby = document.getElementsByName("hobby");
52          for (var i = 0; i < checkHobby.length; i++) {
53              if (checkHobby[i].checked == true)
54                  console.log(checkHobby[i].value + " is checked.");
55              else
56                  console.log(checkHobby[i].value + " is not checked.");
57          }
58      }
59  </script>
60  </html>
```

关于【代码 2-11】的说明：

- 第 16 ~ 21 行代码在表单内定义了一组复选框，用于测试复选框全选、取消、部分选择及判断是否选中的操作。
- 第 29 ~ 59 行代码是对复选框进行全选、取消、部分选择及判断是否选中的操作，主要是通过复选框的“checked”属性来实现的。
  - 第 30 ~ 35 行代码定义的“on_checkbox_all_click()”方法，是实现复选框全部选择的方法。
  - 第 36 ~ 41 行代码定义的“on_checkbox_none_click()”方法，是实现复选框全部取消选择的方法。

- 第 42～48 行代码定义的“on_checkbox_sel_click()”方法，是实现复选框部分选择的方法。其中，第 44 行代码定义了一个布尔型数组，用于存放复选框项是否被选中的状态。
- 第 49～58 行代码定义的“on_checkbox_checked_click()”方法，是实现判断某个复选框是否被选择的方法。

下面使用 Firefox 浏览器运行测试该 HTML 网页，复选框部分选择和判断是否判断是否被选中的效果如图 2.12 所示。

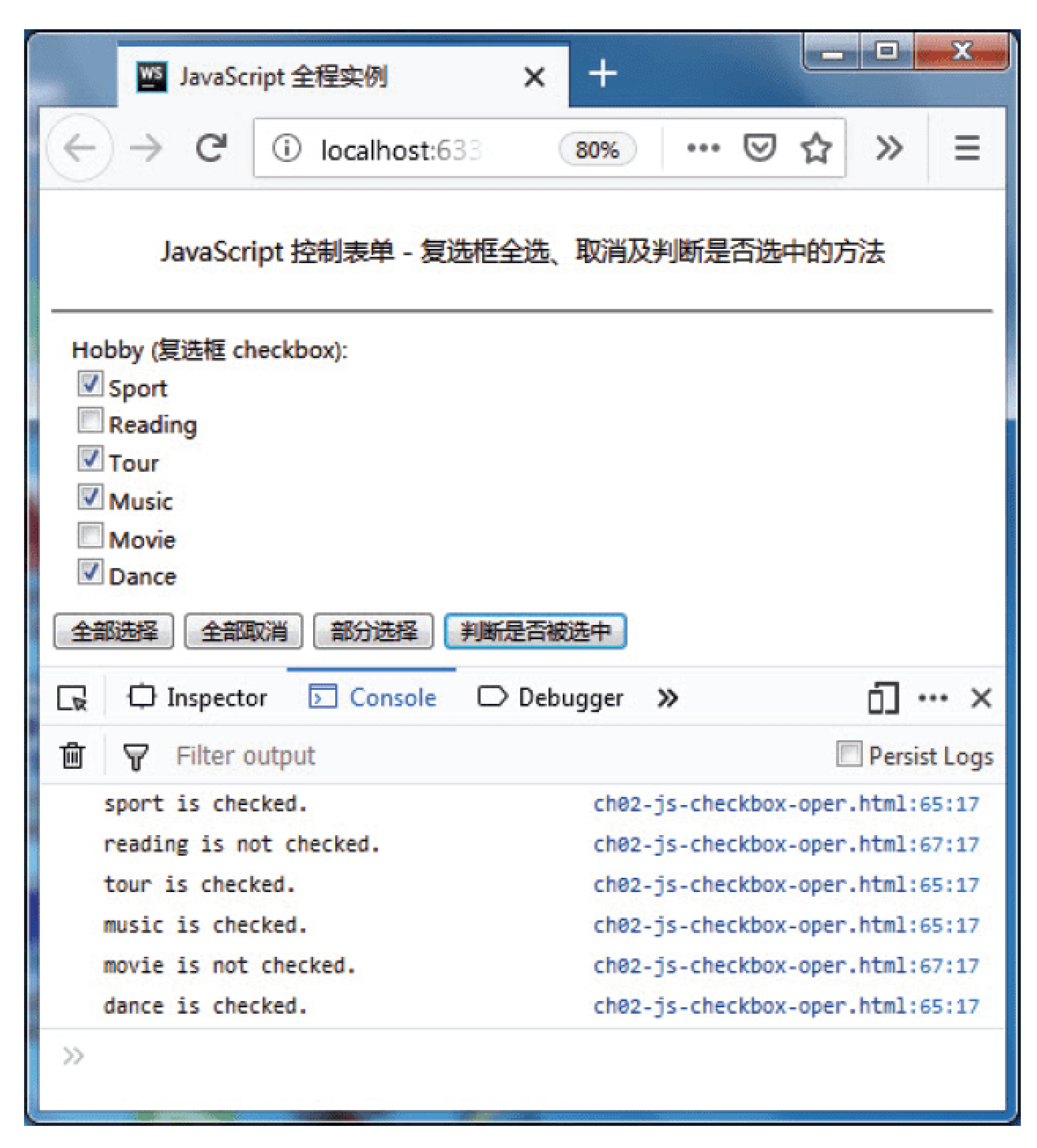


图 2.12　JavaScript 复选框全选、取消及判断是否选中的方法

2.12　如何使用隐藏控件

在 HTML 表单<form>中还可以定义一类隐藏控件（type=hidden），这类控件在页面中是不显示（隐藏）的，但可以实现一些比较特殊的功能。下面介绍一个通过使用隐藏表单控件实现根据用户年龄自动分配卧铺铺位的代码实例。

【代码 2-12】（详见源代码目录 ch02-js-hidden-ele.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
```

```
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 控制表单 - 如何使用隐藏控件</nav>
</header>
<hr>
<!-- 添加文档主体内容 -->
<form name="formHidden" method="get" action="ch02-js-hidden-ele.php">
    <table>
        <caption>乘客卧铺铺位分配</caption>
        <tr>
            <th>姓名：</th>
            <td><input type="text" name="name" id="id-name"/></td>
        </tr>
        <tr>
            <th>出生年月：</th>
            <td>
    <input type="month" name="birth" id="id-birth"
            onchange="on_birth_change(this.id);"/>
                <input type="hidden" name="hidden-level"
                         id="id-hidden-level" value=""/>
            </td>
        </tr>
        <tr>
            <th></th>
            <td><input type="submit" id="id-submit" value="提交"/></td>
        </tr>
    </table>
</form>
</body>
<script type="text/javascript">
    function on_birth_change(thisid) {
        var level = 3;
        var birth = document.getElementById(thisid).value;
        var arrBirth = birth.split('-');
        var birthYear = arrBirth[0];
        var birthMonth = arrBirth[1];
        var now = new Date();
        var nowYear = now.getFullYear();
        var nowMonth = now.getMonth() + 1;
        if ((nowYear - birthYear) > 60) {
            level = 1;
```

```
47          } else if ((nowYear - birthYear) == 60) {
48              if(nowMonth >= birthMonth)
49                  level = 1;
50              else
51                  level = 2;
52          } else if ((nowYear - birthYear) > 45 && (nowYear - birthYear) < 60) {
53              level = 2;
54          } else if ((nowYear - birthYear) == 45) {
55              if(nowMonth >= birthMonth)
56                  level = 2;
57              else
58                  level = 3;
59          } else {
60              level = 3;
61          }
62          document.getElementById("id-hidden-level").value = level;
63      }
64  </script>
65  </html>
```

关于【代码 2-12】的说明：

- 第 14 ~ 33 行代码定义了一个表单，其中 action 属性定义了表单提交的服务器端地址（PHP 文件）。
- 第 24 行代码定义了一个日期类型的文本框，用于输入用户出生年月；同时，还定义了 onchange 事件处理方法（on_birth_change()），当用户选择好日期（日期数据发生改变）后会被触发。
- 第 25 行代码定义了一个隐藏文本框控件（<input type="hidden"/>），用于保存根据用户出生年月计算得出的年龄等级（具体内容见下面的 js 代码）。
- 第 36 ~ 63 行代码是 “on_birth_change()” 方法的实现，具体就是根据第 24 行代码用户选择的出生年月信息和当前的年月信息计算出用户的年龄；如果年龄大于 60 岁（含 60 岁）则自动分配下铺（通过定义 level=1 实现），如果年龄年龄介于 45 ~ 60 岁之间则自动分配中铺（通过定义 level=2 实现），而如果年龄小于 45 岁则自动分配上铺（通过定义 level=3 实现）；第 62 行代码则将 level 等级数值保存在第 25 行代码定义的隐藏文本框中（相当于保存在全局变量中）。表单在提交到服务器端后，服务器端 PHP 文件会根据第 25 行代码定义的隐藏文本框控件中储存的 level 数值获取到用户的年龄等级并自动分配卧铺铺位。

下面使用 Chrome 浏览器运行测试该 HTML 网页，具体效果如图 2.13 所示。



图 2.13　JavaScript 使用隐藏控件的方法

2.13　简单的数字及字符操作

在日常的程序中，经常会涉及数值与字符的操作，例如将一些英文标题转换为大写、将字符型转换为数值型等。本节介绍一些简单的处理方法，读者可以拿来即用。

【代码 2-13】（详见源代码目录 ch02-js-simple-num-char.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<script type="text/javascript">
    window.onload = function () {
        getLowerCase = function (str, type) {//小写转大写
            type = type || "locale";//是否采取本地转换格式
            return type === "locale" && str.toLocaleUpperCase() ||
                     str.toUpperCase();
        };
        getNumbers = function (s) {//字符型转换成数值型
            var n = parseInt(s, 10);//获取转换值
            if (isNaN(n)) return 0;//如果为 NaN 则转换结果为 0
            return n;
        };
        getString = function (n) {  //数字转换成字符型
            return n.toString();
```

```
20          };
21          console.log(getLowerCase("ASDsssss2asdA"));  //小写转大写
22          /*以下是字符转换数字*/
23          console.log(getNumbers("ASDsssss2asdA"));
24          console.log(getNumbers("123123ASDsssss2asdA"));
25          console.log(getNumbers("ASDsssss2asdA123"));
26          /*以下是数字转换成字符型*/
27          console.log(getString(123123) + "类型: " + typeof getString(123123));
28          console.log(getString(234324) + "类型: " + typeof getString(234324));
29      };
30  </script>
31  <body>
32  <!-- 添加文档主体内容 -->
33  <header>
34      <nav>JavaScript 控制表单 - 简单的数字及字符操作</nav>
35  </header>
36  <hr>
37  </body>
38  </html>
```

关于【代码 2-13】的说明：

- 本例一共介绍了 3 个处理函数：小写转大写、字符转数字、数字转字符。
- 第 09～12 行代码是小写转大写，首先会检测 type 参数，然后根据参数来选择相应的内置函数。
- 第 13～17 行代码是将字符转换成数字，调用 JavaScript 脚本语言中的 parseInt()函数进行转换，当被转换值为 NaN 时返回值为 0。
- 第 18～20 行代码是将数字转换成字符，调用 JavaScript 脚本语言中的 toString()函数进行转换。
- 第 21～28 行代码分别对小写转大写、字符转数字、数字转字符这 3 个函数方法进行了测试。

下面使用 FireFox 浏览器运行测试该 HTML 网页，具体效果如图 2.14 所示。



图 2.14　JavaScript 简单的数字及字符操作

2.14　高亮显示表单中的焦点控件

在某些设计场景下，需要将表单中的焦点控件进行高亮显示。下面就介绍一个通过 JavaScript 实现高亮显示表单中焦点控件的代码实例。

【代码 2-14】（详见源代码目录 ch02-js-focus-ele-highlight.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 控制表单 - 高亮显示表单中的焦点控件</nav>
</header>
<!-- 添加文档主体内容 -->
<form name="formHighlight" method="get">
    <table>
        <tr>
        <th>First Name: </th>
        <td><input type="text" id="id-fname" onfocus="highlight(this.id);"
                   value="King"/></td>
        </tr>
        <tr>
        <th>Last Name: </th>
       <td><input type="text" id="id-lname" onfocus="highlight(this.id);"
                  value="Wang"/></td>
        </tr>
        <tr>
        <th></th>
        <td><input type="button" id="id-btn-focus-ele" onfocus=
                   "highlight(this.id);" value="高亮显示焦点控件"/></td>
        </tr>
    </table>
</form>
</body>
<script type="text/javascript">
    function highlight(thisid) {
```

```
32         var frmEle = formHighlight.elements;
33         var eleFocus = document.activeElement;
34         for (var i=0; i<frmEle.length; i++) {
35             if(frmEle[i].id == eleFocus.id) {
36                 console.log(frmEle[i].id);
37                 console.log(eleFocus.id);
38                 document.getElementById(thisid).style.border =
                           "2px solid #666";
39                 document.getElementById(thisid).style.fontWeight = "bold";
40                 document.getElementById(thisid).style.color = "#fff";
41                 document.getElementById(thisid).style.backgroundColor =
                           "#888";
42             } else {
43                 console.log(frmEle[i].id);
44                 console.log(eleFocus.id);
45                 document.getElementById(frmEle[i].id).style.border = "";
46                 document.getElementById(frmEle[i].id).style.fontWeight = "";
47                 document.getElementById(frmEle[i].id).style.color = "";
48                 document.getElementById(frmEle[i].id).
                                           style.backgroundColor = "";
49             }
50         }
51     }
52 </script>
53 </html>
```

关于【代码 2-14】的说明：

- 第 13～28 行代码定义了一个表单，其中定义了一组文本框和一个按钮控件；同时，为这些表单控件定义了 onfocus 事件处理方法（highlight()）。
- 第 31～51 行代码是 highlight()方法的实现，具体就是使用前面实例中介绍过的 activeElement 属性获取当前焦点控件，然后为其添加高亮显示的 CSS 样式代码。

下面使用 FireFox 浏览器运行测试该 HTML 网页，具体效果如图 2.15 所示。



图 2.15　高亮显示表单中的焦点控件

如图 2.15 所示，但用户使用 Tab 键切换表单中控件的焦点时，获取焦点的控件具有了高亮显示的样式。

2.15　动态添加、删除下拉菜单选项

在实际的网站运营中很多数据都是动态变化的，比如一些下拉菜单的选项，此时就需要利用 JavaScript 来动态地添加或删除选项了。其实，所谓的动态添加或删除下拉菜单的选项，实际上就是动态地操作 DOM 对象。JavaScript 提供了为下拉菜单执行添加选项和删除选项的 API —— add()方法和 remove()方法。

【代码 2-15】（详见源代码目录 ch02-js-select-add-del.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 控制表单 - 动态添加、删除下拉菜单选项</nav>
</header>
<!-- 添加文档主体内容 -->
<form name="formSelectAddRemove" method="get">
    <table>
        <caption></caption>
        <tr>
            <th rowspan="1" colspan="1"></th>
            <td rowspan="1" colspan="3">
                <fieldset>
                    <legend>添加选项</legend>
                    <table>
                        <tr>
                            <th rowspan="1" colspan="1">Value:</th>
                            <td rowspan="1" colspan="1">
                <input type="text" class="input-sel" name="optValue"
                        id="id-optValue"/>
                            </td>
                        </tr>
                        <tr>
                            <th rowspan="1" colspan="1">Text:</th>
                            <td rowspan="1" colspan="1">
                <input type="text" class="input-sel" name="optText"
```

```
                              id="id-optText"/></td>
                            </tr>
                            <tr>
                               <th></th>
                               <td rowspan="1" colspan="1">
                               <input type="button" id="id-input-add-opt"
                                      value="添加选项"
                                        onclick="on_add_opt_click(this.id);"/>
                                                </td>
                            </tr>
                        </table>
                    </fieldset>
                </td>
            </tr>
            <tr>
                <th rowspan="1" colspan="1">Hobby:</th>
                <td rowspan="1" colspan="1">
                    <select name="selHobby" id="id-selHobby">
                        <option value="">请选择...</option>
                        <option value="sport" selected="selected">Sport</option>
                        <option value="reading">Reading</option>
                        <option value="tour">Tour</option>
                    </select>
                </td>
                <th></th>
                <td rowspan="1" colspan="1">
                    <input type="button" id="id-input-remove-opt"
                           value="删除选项"
                          onclick="on_remove_opt_click(this.id);"/>
                </td>
            </tr>
        </table>
    </form>
</body>
<script type="text/javascript">
    /**
     * remove option
     * @param thisid
     */
    function on_remove_opt_click(thisid) {
        var selHobby = document.getElementById("id-selHobby");
        selHobby.remove(selHobby.selectedIndex);
    }
```

```
71        /**
72         * add option
73         * @param thisid
74         */
75        function on_add_opt_click(thisid) {
76            var selHobby = document.getElementById("id-selHobby");
77            var option = document.createElement("option");
78            option.value = document.getElementById("id-optValue").value;
79            option.text = document.getElementById("id-optText").value;
80            selHobby.add(option, null);
81        }
82    </script>
83    </html>
```

关于【代码 2-15】的说明：

- 第 46～51 行代码定义了一个下拉菜单，并初始化了几个选项；然后动态添加和删除选项的操作都是基于这个下拉菜单来操作的。
- 第 55～56 行代码定义了一个按钮控件，并定义了 onclick 单击事件方法（on_remove_opt_click()），用于执行删除选项的操作。第 67～70 行代码是事件方法（on_remove_opt_click()）的具体实现，第 69 行代码通过 remove()方法删除指定的选项（selectedIndex）。
- 第 36～37 行代码定义了另一个按钮控件，并定义了 onclick 单击事件方法（on_add_opt_click()），用于执行添加选项的操作。第 75～81 行代码是事件方法（on_add_opt_click()）的具体实现，第 78～79 行代码获取了用户输入的选项信息（value 值和 text 文本），第 80 行代码通过 add()方法添加刚刚定义的选项。

下面使用 FireFox 浏览器运行测试该 HTML 网页，具体效果如图 2.16 和图 2.17 所示。

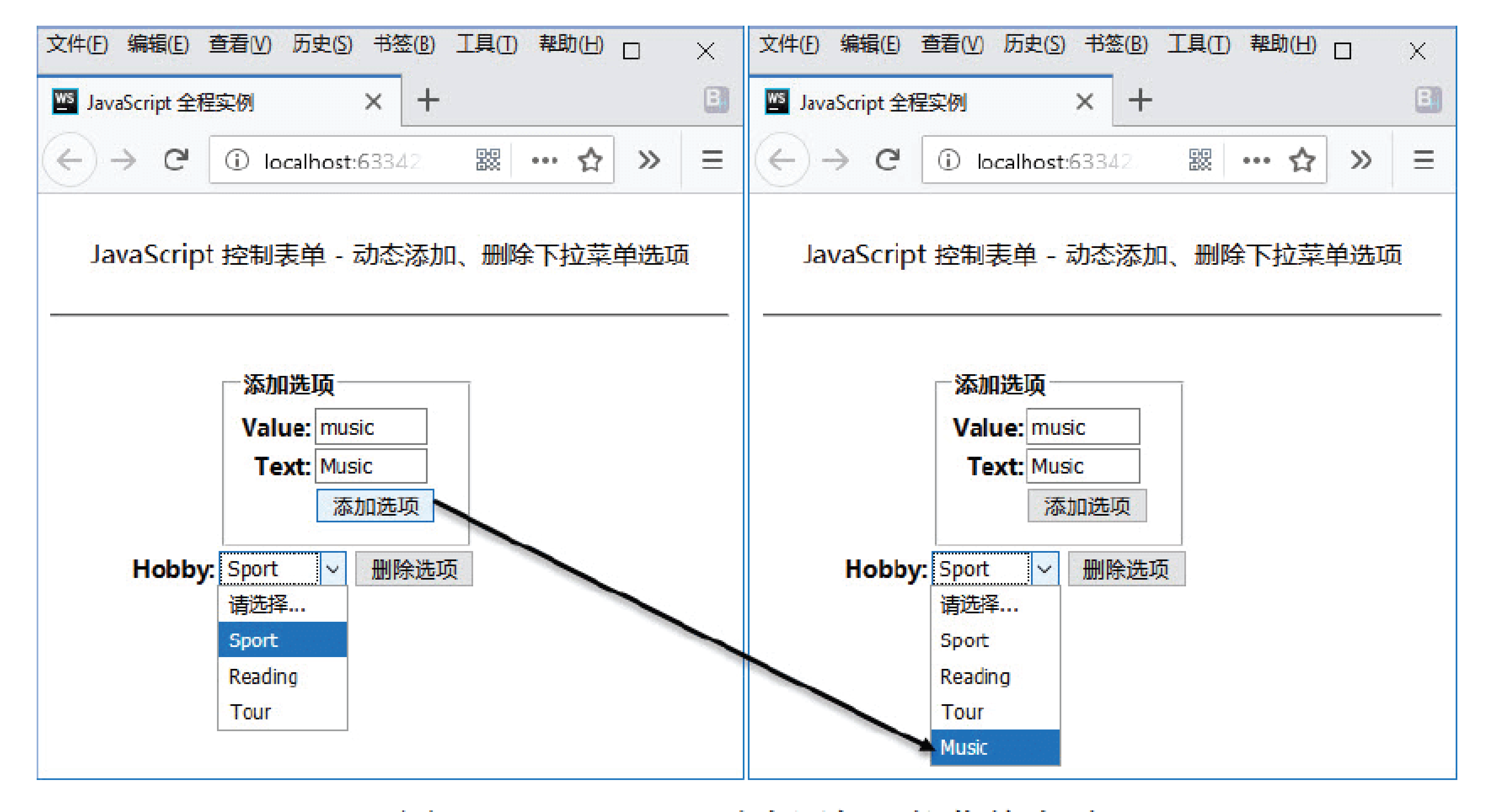


图 2.16　JavaScript 动态添加下拉菜单选项

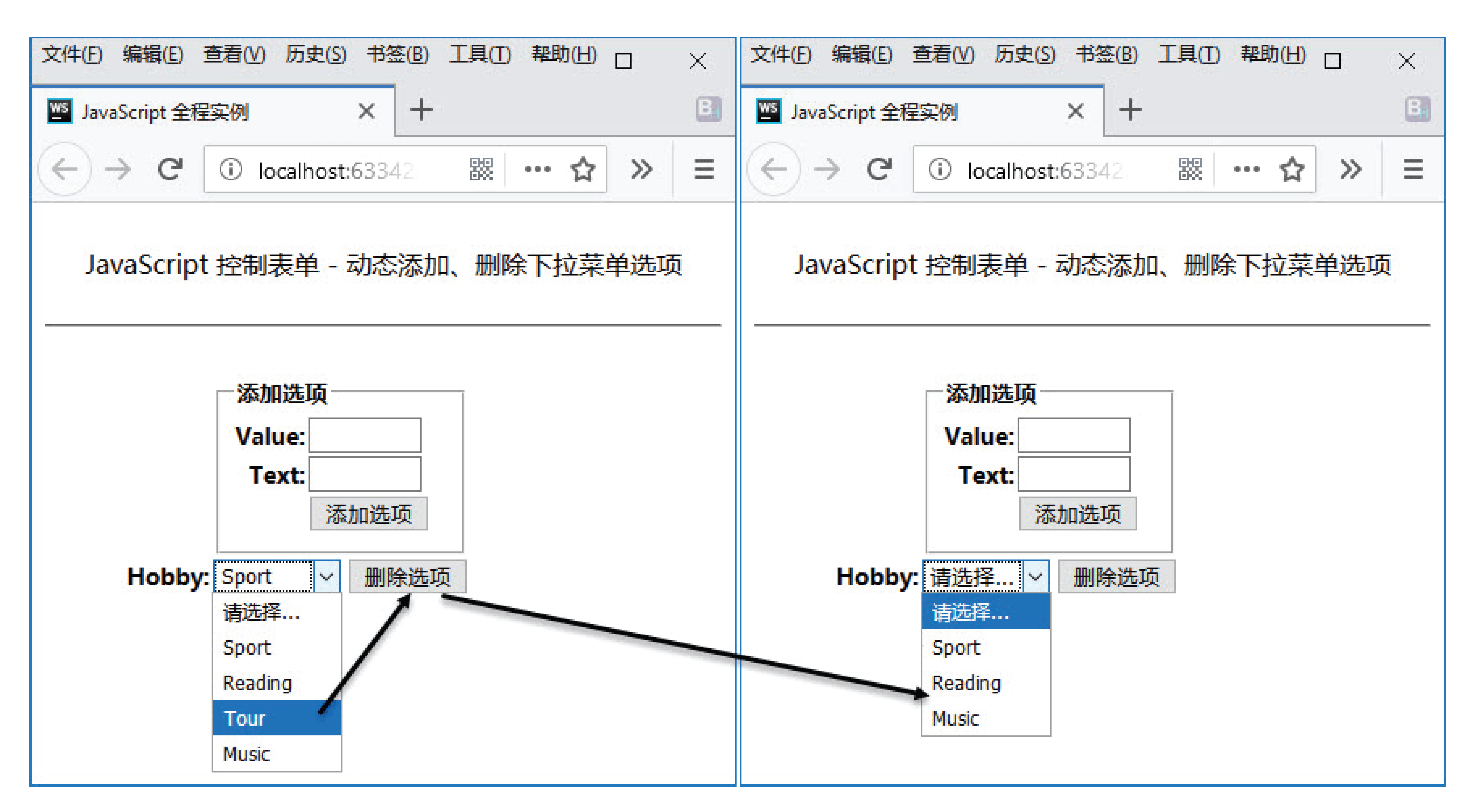


图 2.17　JavaScript 动态删除下拉菜单选项

第 3 章　JavaScript 控制 DOM

本章介绍如何通过 JavaScript 来控制 HTML DOM。DOM 是 HTML 网页的标准模型和编程接口，是网页开发中重要的基础核心。

3.1　JavaScript 与 HTML DOM

DOM（Document Object Model，文档对象模型）是 W3C 组织推荐的处理可扩展标志语言的标准编程接口。在 HTML 网页中，页面的全部对象被组织成一个树形结构，而这个结构模型也就是所谓的 HTML DOM。

HTML DOM 是 W3C 组织推出的标准对象模型和标准编程接口，其定义了所有 HTML 元素的对象和属性，以及访问它们的方法。也就是说，HTML DOM 是关于如何操作（获取、添加、修改或删除）HTML 元素对象的标准。

相信读者都听说过 HTML 网页有静态页面和动态页面之分。如果一个网页中仅仅是由各种 HTML 标签元素组合而成，那么一般称之为静态页面。如果在一个网页中使用了 JavaScript 脚本代码，就可以称之为动态页面了。JavaScript 就是通过操作 HTML DOM 对象模型实现了对 HTML 页面中的元素、属性、样式和事件的控制，从而构建出 HTML 动态页面。

HTML DOM 定义了一个 Document 对象（每一个载入浏览器的 HTML 文档都会生成一个 Document 对象），通过该对象可以使用 JavaScript 脚本语言对 HTML 页面中的元素对象进行访问。下面给出一个常用的、通过元素对象的 id 属性和 name 属性获取对该元素对象引用的代码实例。

【代码 3-1】

```
/**
* 通过 id 属性和 name 属性获取对象
*/
document.getElementById(id)           // TODO: 通过 id 获取对象
document.getElementsByName(name)      // TODO: 通过 name 获取对象
```

3.2　通过 id 获取网页中的元素对象

HTML DOM 定义了一个 getElementById()方法，专门用于获取对拥有指定 id 的元素对象集合中的第 1 个元素对象的引用。下面介绍一个 JavaScript 通过 id 属性获取对元素对象引用的代码实例。

【代码 3-2】（详见源代码目录 ch03-js-get-id.html 文件）

```
01 <!doctype html>
02 <html lang="en">
03 <head>
04   <!-- 添加文档头部内容 -->
05   <title>JavaScript 全程实例</title>
06 </head>
07 <body>
08   <!-- 添加文档主体内容 -->
09   <header>
10       <nav>JavaScript 控制 DOM - 通过 id 获取网页中的元素对象</nav>
11   </header>
12   <!-- 添加文档主体内容 -->
13   <p>First Name: <input type="text" name="fname" id="id-fname" /></p>
14   <p>Last Name: <input type="text" name="lname" id="id-lname" /></p>
15   <input type="button" id="id-get-id" onclick="on_get_id();"
            value="通过 id 获取" />
16   <div>
17       <span id="id-span-1">Full Name :  </span><br>
18       <span id="id-span-2">Full Name :  </span><br>
19       <span id="id-span-3">Full Name :  </span><br>
20   </div>
21 </body>
22 <script type="text/javascript">
23   function on_get_id() {
24        var id_fname = document.getElementById("id-fname");
25        var id_lname = document.getElementById("id-lname");
26        var full_name = id_fname.value + " " + id_lname.value;
27        for(var i=1; i<=3; i++)
28            document.getElementById("id-span-" + i).innerText += full_name;
29   }
30 </script>
31 </html>
```

关于【代码 3-2】的说明：

- 第 13～14 行代码通过标签<input>定义一组文本框（分别定义了 id 属性），用于输入姓名。
- 第 17～19 行代码通过标签<span>定义了一组行内元素（分别定义了有一定规则的 id 属性），用于显示从第 13～14 行代码中定义的文本框中获取姓名。
- 第 27～28 行代码通过一个 for 语句循环调用 getElementById()方法，依次将姓名信息显示在第 17～19 行代码定义的<span>行内标签元素中。这里特别需要留意的是，getElementById()方法中 id 参数是根据自变量 i 的取值来生成的。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 3.1 所示。

图 3.1　JavaScript 通过 id 获取网页中的元素对象

3.3　通过 name 获取网页中的复选框

HTML DOM 还定义了一个 getElementsByName()方法，专门用于获取对拥有指定 name 的全部元素对象集合的引用。下面介绍一个 JavaScript 通过 name 属性获取复选框（Checkbox）对象集合引用的代码实例。

【代码 3-3】（详见源代码目录 ch03-js-get-checkbox.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 控制 DOM - 通过 name 获取网页中的复选框</nav>
</header>
<!-- 添加文档主体内容 -->
<p>Hobby: <br>
```

```
    <input type="checkbox" name="hobby" value="sport" checked />Sport
    <input type="checkbox" name="hobby" value="reading"/>Reading
    <input type="checkbox" name="hobby" value="tour" checked />Tour
    <input type="checkbox" name="hobby" value="music"/>Music&
    <input type="checkbox" name="hobby" value="movie" checked />Movie
    <input type="checkbox" name="hobby" value="dance"/>Dance
</p>
<p>
    <input type="button" onclick="on_get_checkbox();" value="获取全部"/>
    <input type="button" onclick="on_get_checkbox_checked();"
           value="获取全部选中项"/>
  <input type="button" onclick="on_get_checkbox_unchecked();"
         value="获取全部非选中项"/>
</p>
</body>
<script type="text/javascript">
    /**
     * Func - Get all Checkbox
     */
    function on_get_checkbox() {
        var cb = document.getElementsByName("hobby");
        console.log("Checkbox --- ");
        cb.forEach(function (item) {
        console.log("value : " + item.value + ", text : " +
                    item.nextSibling.nodeValue);
        });
    }
    /**
     * Func - Get all checked Checkbox
     */
    function on_get_checkbox_checked() {
        var cb = document.getElementsByName("hobby");
        console.log("checked Checkbox --- ");
        cb.forEach(function (item) {
            if (item.checked)
            console.log("value : " + item.value + ", text : " +
                        item.nextSibling.nodeValue);
        });
    }
    /**
     * Func - Get all unchecked Checkbox
     */
```

```
52      function on_get_checkbox_unchecked() {
53          var cb = document.getElementsByName("hobby");
54          console.log("unchecked Checkbox --- ");
55          cb.forEach(function (item) {
56              if (!item.checked)
57              console.log("value : " + item.value + ", text : " +
                            item.nextSibling.nodeValue);
58          });
59      }
60  </script>
61  </html>
```

关于【代码 3-3】的说明：

- 第 14 ~ 19 行代码通过标签<input>定义一组复选框（定义了相同的 name 属性），表明是一个分组的复选框。
- 第 32 行代码通过 name 属性获取了全部复选框的对象集合，第 36 ~ 38 行代码通过 forEach 迭代语句遍历了全部复选框，并通过浏览器控制台输出了复选框的 value 属性值。
- 第 45 行和第 56 行代码分别通过判断复选框的 checked 属性来筛选当前复选框是否被选中，然后通过浏览器控制台输出了复选框的 value 属性值。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 3.2 和图 3.3 所示。

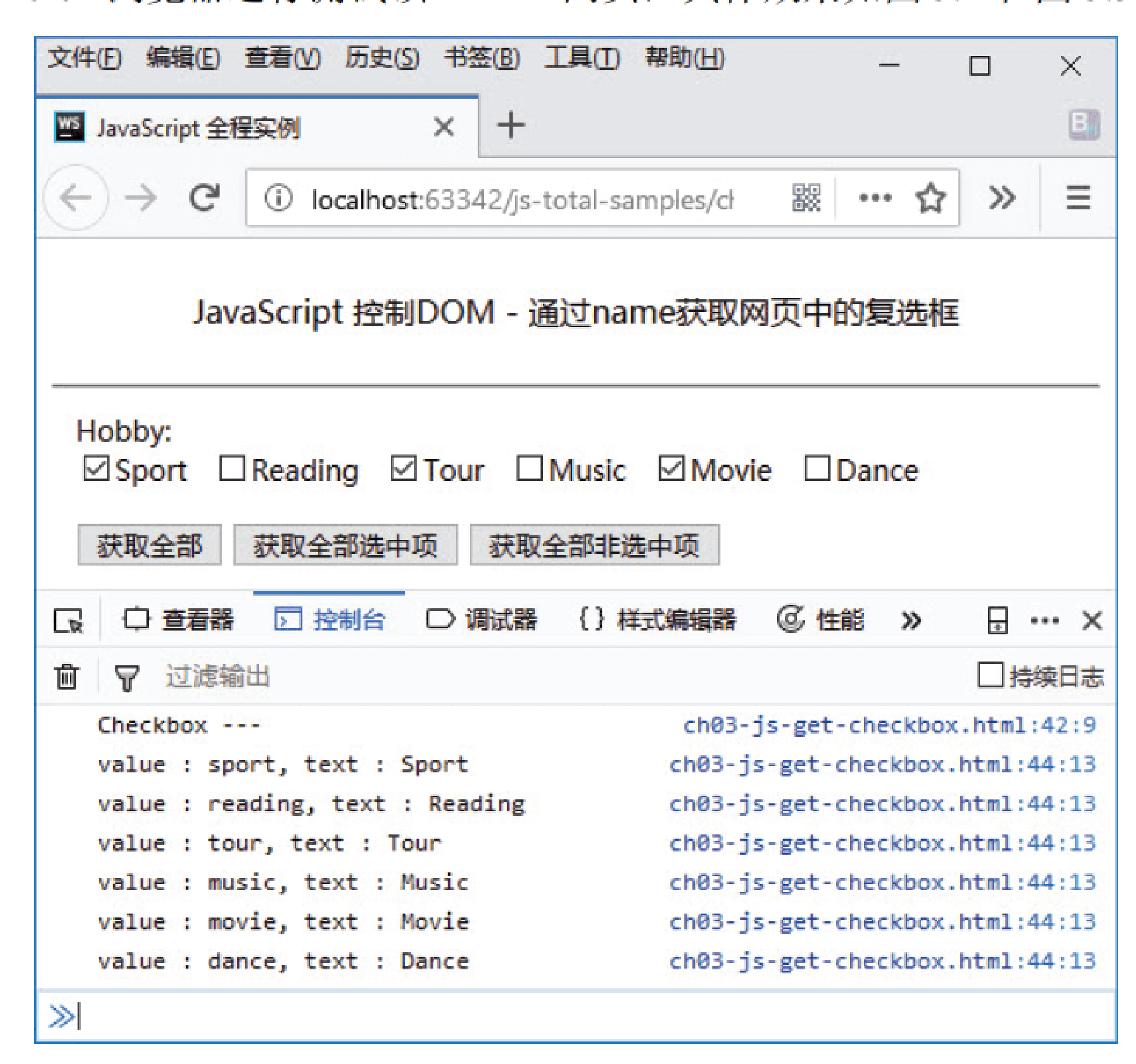


图 3.2　JavaScript 通过 name 属性获取网页中的全部复选框

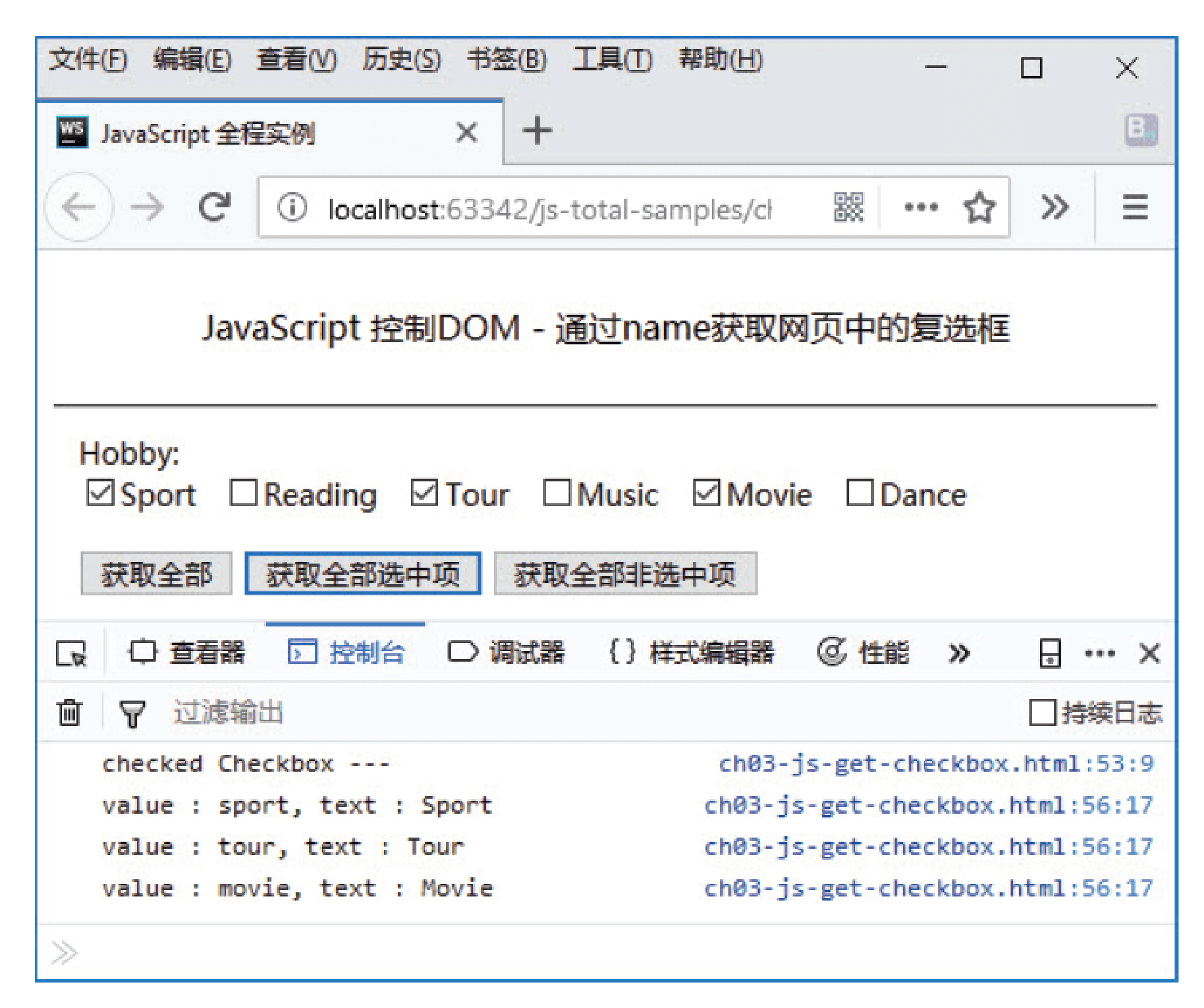


图 3.3　JavaScript 通过 name 属性获取网页中全部选中的复选框

3.4　通过标签名获取网页中的多个文本

除了前面介绍的通过标签元素的 id 属性和 name 属性获取对象引用，HTML DOM 还定义了一个 getElementsByTagName()方法，专门用于获取对拥有指定标签名的全部元素对象集合的引用。

下面介绍一个 JavaScript 通过标签名获取网页中多个文本的代码实例。

【代码 3-4】（详见源代码目录 ch03-js-get-tag.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 控制 DOM - 通过标签名获取网页中的多个文本</nav>
</header>
<!-- 添加文档主体内容 -->
<p>First Name: <input type="text" name="fname" id="id-fname"/></p>
<p>Last Name: <input type="text" name="lname" id="id-lname"/></p>
<input type="button" onclick="on_get_tag_text();" value="获取(text)文本"/>
```

```
16 <input type="button" onclick="on_get_tag_button();"
          value="获取(button)文本"/>
17 </body>
18 <script type="text/javascript">
19     /**
20      * Func - get input[type=text]
21      */
22     function on_get_tag_text() {
23         var inputs = document.getElementsByTagName("input");
24         console.log("input[type=text] --- ");
25         for (var i = 0; i < inputs.length; i++) {
26             if (inputs[i].type == "text")
27                 console.log("value : " + inputs[i].value);
28         }
29     }
30     /**
31      * Func - get input[type=button]
32      */
33     function on_get_tag_button() {
34         var inputs = document.getElementsByTagName("input");
35         console.log("input[type=button] --- ");
36         for (var i = 0; i < inputs.length; i++) {
37             if (inputs[i].type == "button")
38                 console.log("value : " + inputs[i].value);
39         }
40     }
41 </script>
42 </html>
```

关于【代码 3-4】的说明：

- 第 13 ~ 14 行代码通过标签<input type="text">定义一组文本框，第 15 ~ 16 行代码通过标签<input type="button">定义另一组按钮控件，而且这两组均是通过<input>标签来定义的。
- 第 23 行代码通过标签名获取了全部<input>标签元素的对象集合。第 25 ~ 28 行代码通过 for 循环语句遍历了全部<input>标签元素。其中，第 26 行代码通过判断 type 属性值（text）来筛选集合中的文本框类型，并通过浏览器控制台输出了文本框的 value 属性值。
- 第 37 行代码通过判断 type 属性值（button）来筛选集合中的按钮控件类型，并通过浏览器控制台输出了按钮控件的 value 属性值。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 3.4 所示。

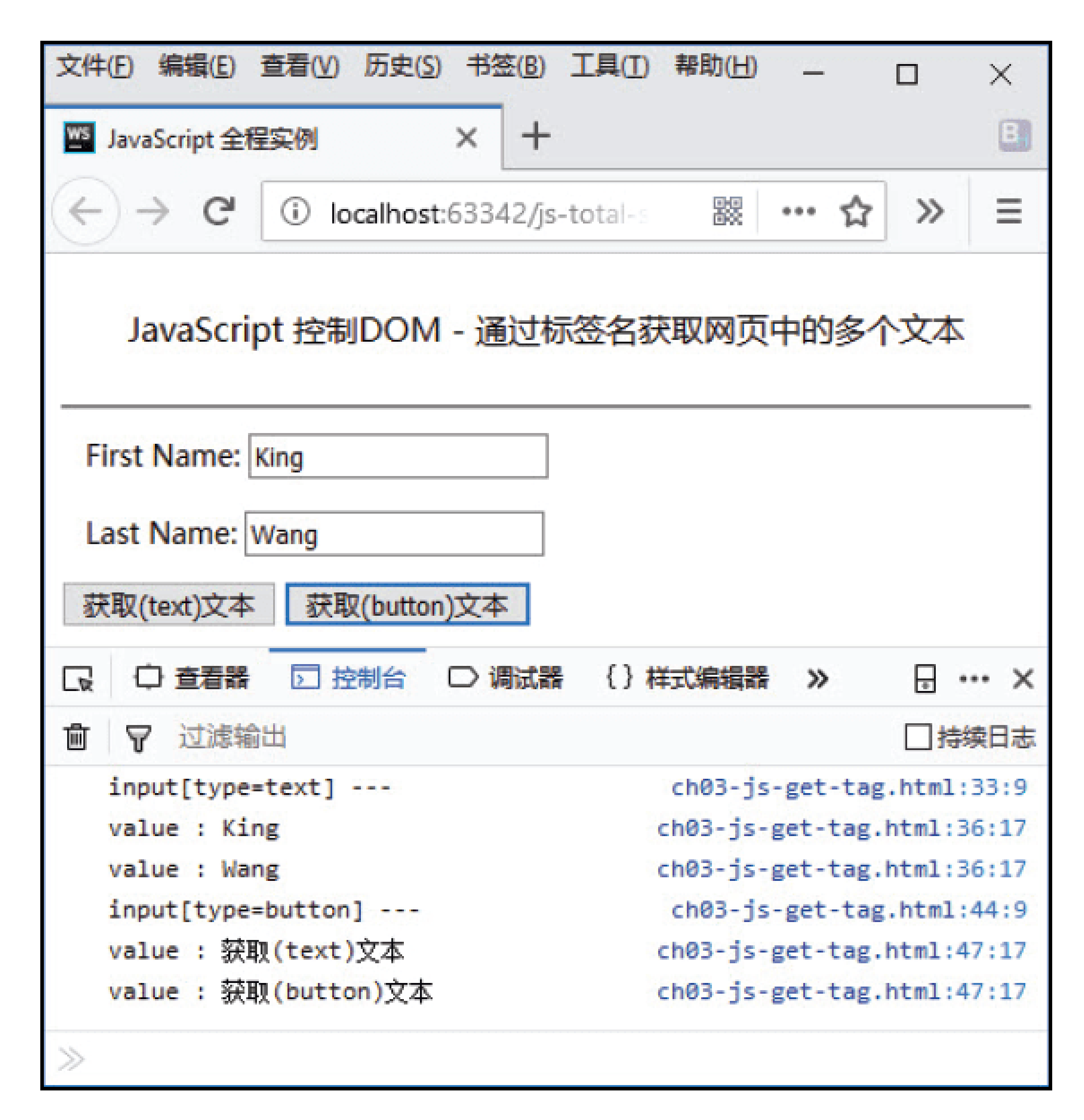


图 3.4　JavaScript 通过标签名获取网页中的多个文本

3.5　遍历网页元素的全部属性

HTML DOM 为标签元素定义了一个 attributes 属性，用于返回指定标签元素的属性集合。借助 attributes 属性，JavaScript 通过一个循环语句就可以实现遍历标签元素全部属性的操作了。

【代码 3-5】（详见源代码目录 ch03-js-get-attributes.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 控制 DOM - 遍历网页元素的全部属性</nav>
</header>
<!-- 添加文档主体内容 -->
<p>Name: <input type="text" class="css" name="name" id="id-name"
                value="King"/></p>
<input type="button" id="id-get-attr" onclick="on_get_attr();"
        value="获取标签的全部属性"/>
</body>
```

```
16 <script type="text/javascript">
17     /**
18      * Func - get attributes
19      */
20     function on_get_attr() {
21         var name = document.getElementById("id-name");
22         var attrs = name.attributes;
23         console.log("input name's all attributes --- ");
24         for (var i = 0; i < attrs.length; i++) {
25             console.log("attr name : " + attrs[i].name + ", value : " +
                               attrs[i].value);
26         }
27     }
28 </script>
29 </html>
```

关于【代码 3-5】的说明：

- 第 13 行代码通过标签<input>定义一个文本框，为了测试 attributes 属性的使用，特意定义了 type、class、name、id（id="id-name"）和 value 这 5 个属性。
- 第 21 行代码通过 id 属性获取了文本框（id="id-name"）对象。
- 第 22 行代码通过 attributes 属性获取了该文本框全部属性的集合（attrs）。
- 第 24～26 行代码通过 for 循环语句遍历了属性集合（attrs）。其中，第 25 行代码通过浏览器控制台输出了文本框各个属性的名称（name）和属性值（value）。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 3.5 所示。

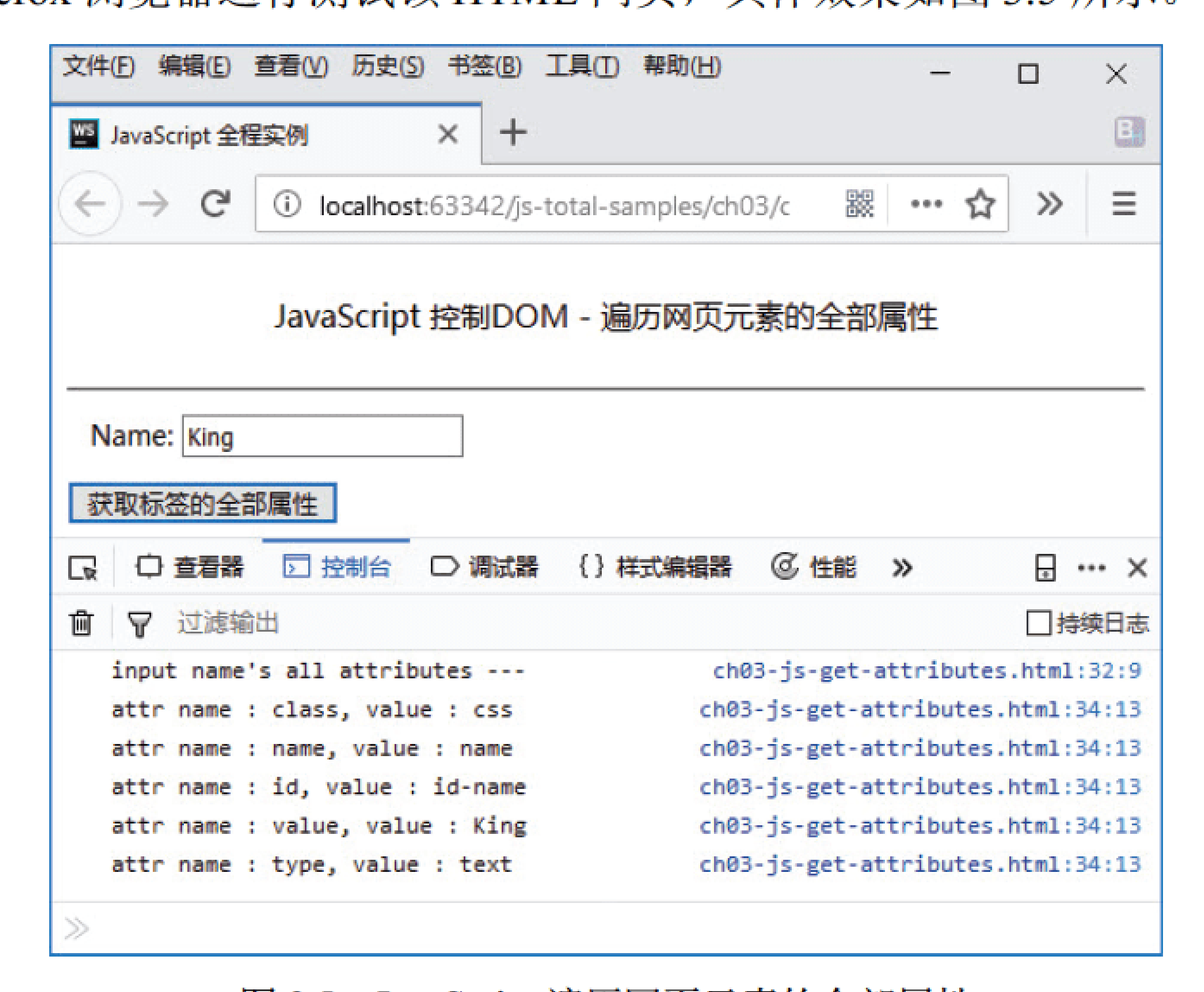


图 3.5　JavaScript 遍历网页元素的全部属性

3.6 动态创建网页新文本段落

HTML DOM 定义了一个 createElement()方法，可以用于创建各种元素节点，同时该方法返回一个元素对象。在新的元素节点创建好后，再通过 appendChild()方法或 insertBefore()方法插入元素节点，就可以实现动态创建网页元素的功能了。

下面介绍一个很常用的 JavaScript 动态创建新的文本段落（<p>标签元素）的代码实例。

【代码 3-6】（详见源代码目录 ch03-js-createElement-p.html 文件）

```
<!doctype html>
<html lang="en">
<head>
  <!-- 添加文档头部内容 -->
  <title>JavaScript 全程实例</title>
</head>
<body>
  <!-- 添加文档主体内容 -->
  <header>
      <nav>JavaScript 控制 DOM - 动态创建网页新文本段落</nav>
  </header>
  <!-- 添加文档主体内容 -->
  <div id="id-div1" class="css-p"></div>
  <div id="id-div2" class="css-p">
      <p>已经存在的文本段落.</p>
  </div>
  <div id="id-div3" class="css-p">
      <p>第一段文本段落.</p>
      <p>第三段文本段落.</p>
  </div>
  <input type="button" onclick="on_createEle_p_new();"
         value="动态创建文本段落" />
  <input type="button" onclick="on_createEle_p_append();"
         value="动态追加文本段落" />
  <input type="button" onclick="on_createEle_p_insert();"
         value="动态插入文本段落" />
</body>
<script type="text/javascript">
   function on_createEle_p_new() {
      var id_div1 = document.getElementById("id-div1");
```

```
28          var p1 = document.createElement("p");
29          p1.innerText = "动态创建的文本段落.";
30          id_div1.appendChild(p1);
31      }
32      function on_createEle_p_append() {
33          var id_div2 = document.getElementById("id-div2");
34          var p2_2 = document.createElement("p");
35          p2_2.innerText = "动态追加(appendChild)的文本段落.";
36          id_div2.appendChild(p2_2);
37      }
38      function on_createEle_p_insert() {
39          var id_div3 = document.getElementById("id-div3");
40          var child_p = id_div3.children;
41          var p3_1 = child_p[0];
42          var p3_3 = child_p[1];
43          var p3_2 = document.createElement("p");
44          p3_2.innerText = "动态插入(insertAfter)的文本段落.";
45          id_div3.insertBefore(p3_2, p3_3);
46      }
47  </script>
48  </html>
```

关于【代码 3-6】的说明：

- 第 13 行、第 14～16 行和第 17～20 行代码分别通过标签<div>定义 3 个层对象，在每个层中分别加入了不同数量的段落标签<p>，目的是为了分别实现动态创建、追加和插入文本段落的功能。
- 第 26～31 行代码定义的函数实现了动态创建文本段落的操作；第 28～29 行代码通过 createElement()方法创建了一个段落<p>对象（p1），并通过 innerText 属性定义了文本内容；第 30 行代码通过 appendChild()方法将新创建的文本段落（p1）对象动态插入到第 13 行代码定义的标签<div>中。
- 第 32～37 行代码定义的函数实现了动态追加文本段落的操作。注意，appendChild()方法会将新对象自动插入同级兄弟对象的最后位置。
- 第 38～46 行代码定义的函数实现了动态插入文本段落的操作。注意，该操作是通过 insertBefore()方法实现的，因为 appendChild()方法是无法实现将新对象插入到同级兄弟对象之间的。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 3.6 所示。

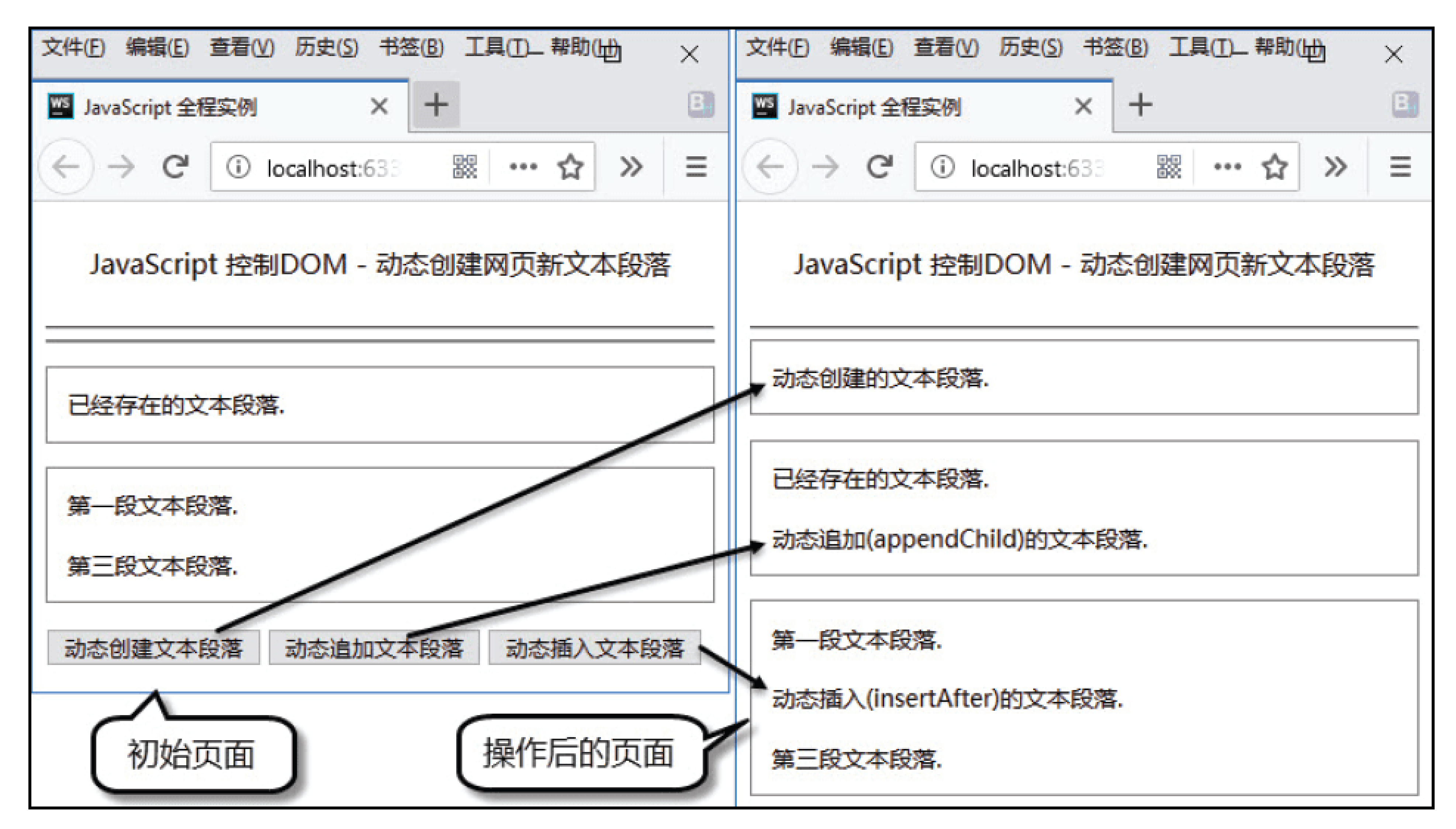


图 3.6　JavaScript 动态创建网页新文本段落

3.7　动态删除网页文本段落

在 3.6 节中介绍了动态创建新文本段落的方法，本节将介绍动态删除文本段落的方法。HTML DOM 对象定义了一组 remove()方法和 removeChild()方法，可以用于动态删除各种元素节点。

下面给出一个 JavaScript 通过 remove()方法和 removeChild()动态删除文本段落（<p>标签元素）的代码实例。

【代码 3-7】（详见源代码目录 ch03-js-remove-p.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 控制 DOM - 动态删除网页文本段落</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div1" class="css-p">
    <p>动态创建的文本段落.</p>
</div>
<div id="id-div2" class="css-p">
```

```
17      <p>已经存在的文本段落.</p>
18      <p>动态追加(appendChild)的文本段落.</p>
19  </div>
20  <div id="id-div3" class="css-p">
21      <p>第一段文本段落.</p>
22      <p>动态插入(insertAfter)的文本段落.</p>
23      <p>第三段文本段落.</p>
24  </div>
25  <input type="button" onclick="on_removeChild_p();" value="动态删除文本段落"/>
26  <input type="button" onclick="on_remove_p();"
            value="动态删除(remove)文本段落"/>
27  <input type="button" onclick="on_removeChild_all_p();"
            value="动态删除全部文本段落"/>
28  </body>
29  <script type="text/javascript">
30      function on_removeChild_p() {
31          var id_div1 = document.getElementById("id-div1");
32          var child_p = id_div1.childNodes;
33          id_div1.removeChild(child_p[0]);
34      }
35      function on_remove_p() {
36          var id_div2 = document.getElementById("id-div2");
37          var child_p = id_div2.childNodes;
38          id_div2.remove(child_p);
39      }
40      function on_removeChild_all_p() {
41          var id_div3 = document.getElementById("id-div3");
42          var child_p = id_div3.childNodes;
43          for (var i = child_p.length - 1; i >= 0; i--) {
44              id_div3.removeChild(child_p[i]);
45          }
46      }
47  </script>
48  </html>
```

关于【代码 3-7】的说明：

- 第 13～24 行代码分别通过标签<div>定义 3 个层对象，并通过手动方式定义多个文本段落，最终显示成如图 3.6 中操作后的页面效果（大致方向就是将图 3.6 中的效果进行逆向操作）。
- 第 30～34 行代码定义的函数实现了动态删除文本段落的操作。其中，第 32 行代码通过 childNodes 属性获取了第一个层（<div id="id-div1">）中的全部段落<p>节点的集合（child_p），第 33 行代码通过 removeChild()方法删除了该文本段落。注意，删除时必须使用数组下标的方式（child_p[0]），而直接删除集合（child_p）对象是无效的。

- 那么能不能直接删除集合（child_p）对象呢？第 35 ~ 39 行代码定义的函数实现了直接删除集合（child_p）对象的操作，不过这里要使用 remove()方法来操作。
- 第 40 ~ 46 行代码定义的函数演示了常规删除多个文本段落的操作，主要就是通过 for 循环语句依次调用 removeChild()方法来删除每一个文本段落。这里要特别注意的是，for 循环语句迭代的方式从大到小，因为如果迭代顺序是从小到大，那么每次删除操作后集合（child_p）对象的数组下标就会发生改变，从而导致删除操作无法完成。

3.8 动态替换段落的文本内容

前面两节分别介绍了动态创建和删除文本段落的方法，本节再介绍一下如何动态替换文本内容的方法。动态替换文本内容主要是通过 innerText 属性或 innerHTML 属性来实现的。其中，innerText 属性主要用来操作纯文本；innerHTML 属性既可以用来操作文本，也可以用来操作 HTML 代码。

下面给出一个 JavaScript 通过 innerText 属性和 innerHTML 属性动态替换文本内容的代码实例。

【代码 3-8】（详见源代码目录 ch03-js-replace-p.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 控制 DOM - 动态替换段落的文本内容</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div" class="css-p">
    <p>文本段落的原始内容.</p>
    <p>文本段落的原始内容.</p>
</div>
<input type="button" id="id-replace-p" onclick="on_replace_p();"
        value="动态替换文本内容"/>
</body>
<script type="text/javascript">
    function on_replace_p() {
        var id_p = document.getElementsByTagName("p");
```

```
22          id_p[0].innerText = "文本段落动态替换的内容.";
23          id_p[1].innerHTML = "<b>文本段落动态替换的内容.</b>";
24      }
25  </script>
26  </html>
```

关于【代码 3-8】的说明：

- 第 22 行代码通过使用 innerText 属性动态替换了第 14 行代码中定义的段落中的文本内容。
- 第 23 行代码通过使用 innerHTML 属性动态替换了第 15 行代码中定义的段落中的文本内容。注意，这里在使用 innerHTML 属性时加入了 HTML 代码（<b>）实现字体加粗的效果。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 3.7 所示。

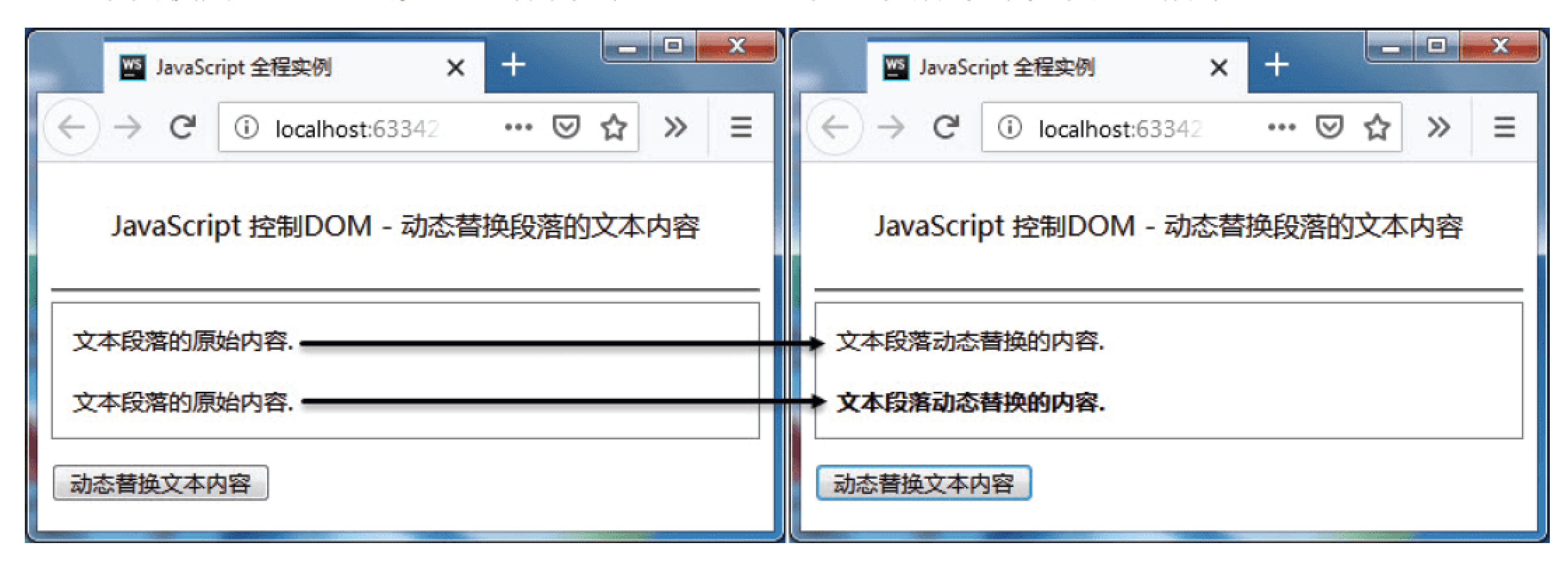


图 3.7　JavaScript 动态替换段落的文本内容

3.9　如何主动触发按钮单击事件

众所周知，页面中设计的按钮控件是为了让用户单击来触发操作的，那么能不能直接通过 JavaScript 脚本语言从后台主动触发按钮来操作呢？答案是肯定的。

下面实现一个 JavaScript 主动触发按钮单击事件的代码实例。

【代码 3-9】（详见源代码目录 ch03-js-btn-click.html 文件）

```
<!doctype html>
<html lang="en">
<head>
  <!-- 添加文档头部内容 -->
  <title>JavaScript 全程实例</title>
</head>
<body>
```

```
08    <!-- 添加文档主体内容 -->
09    <header>
10        <nav>JavaScript 控制 DOM - 主动触发按钮单击事件</nav>
11    </header>
12    <!-- 添加文档主体内容 -->
13    <p>First Name: <input type="text" name="fname" id="id-fname"
                            value="King" /></p>
14    <p>Last Name: <input type="text" name="lname" id="id-lname"
                            value="Wang" /></p>
15    <input type="button" id="id-btn-name" value="用户触发按钮" />
16    <input type="button" id="id-btn-click" value="主动触发按钮" />
17  </body>
18  <script type="text/javascript">
19     var v_btn_name = document.getElementById("id-btn-name");
20     v_btn_name.onclick = function () {
21         var id_fname = document.getElementById("id-fname");
22         var id_lname = document.getElementById("id-lname");
23         var full_name = id_fname.value + " " + id_lname.value;
24         console.log("Full Name : " + full_name);
25     };
26     var v_btn_click = document.getElementById("id-btn-click");
27     v_btn_click.onclick = function () {
28         console.log("主动触发按钮 : ");
29         document.getElementById("id-btn-name").click();
30     };
31  </script>
32  </html>
```

关于【代码 3-9】的说明：

- 第 15 行代码通过标签<input>定义了第一个按钮控件；第 20～25 行代码为该按钮控件添加了 onclick 事件，获取了第 13～14 行代码定义的文本框中的内容。注意，该按钮的单击事件是必须由用户执行触发操作来完成的。
- 第 16 行代码通过标签<input>定义了第二个按钮控件；第 27～30 行代码为该按钮控件添加了 onclick 事件，其中第 29 行代码通过为第一个按钮控件执行 click()方法来执行单击操作。注意，这里第一个按钮控件的单击操作是 JavaScript 主动触发的，而不是由用户被动触发的。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 3.8 和图 3.9 所示。

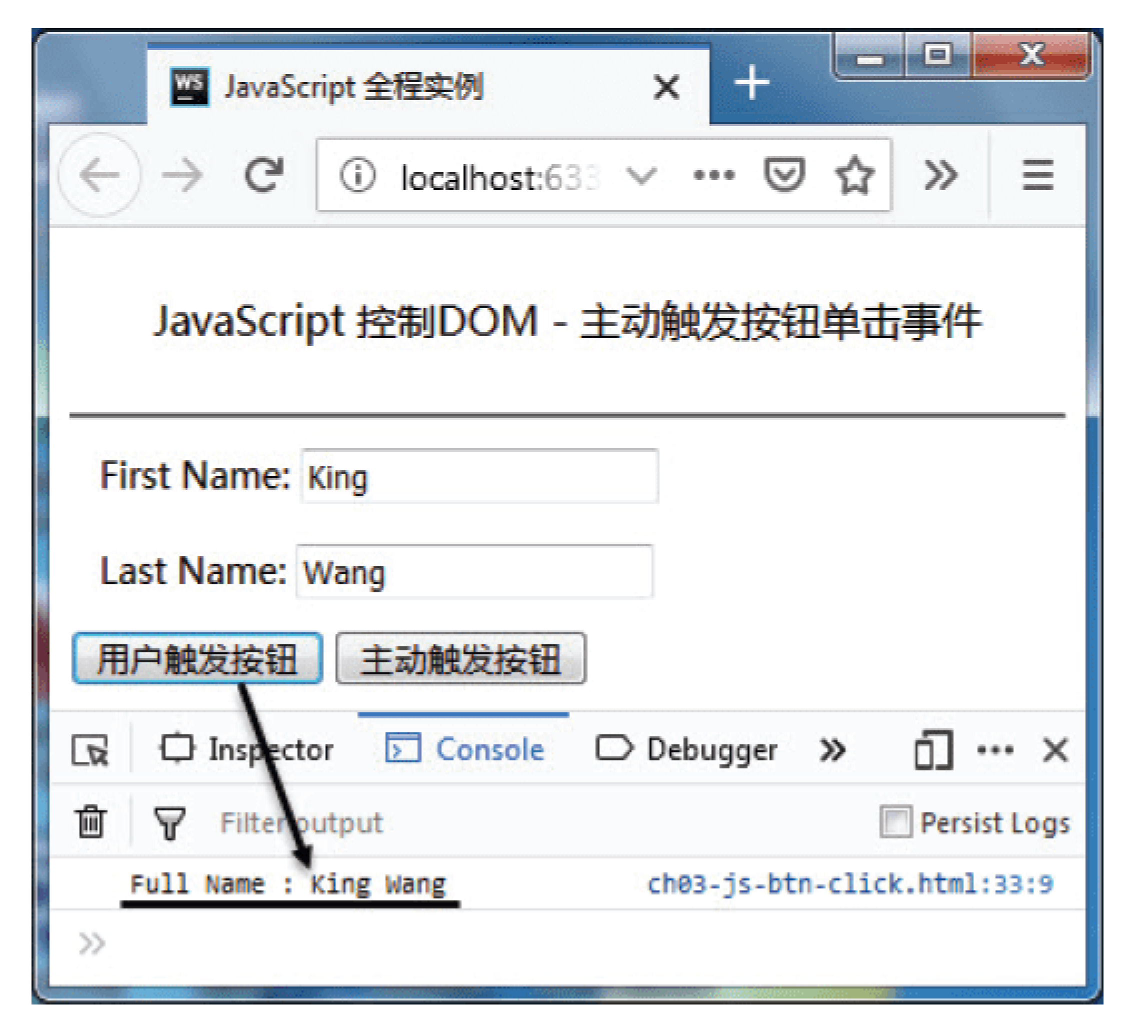


图 3.8　JavaScript 用户触发按钮单击事件

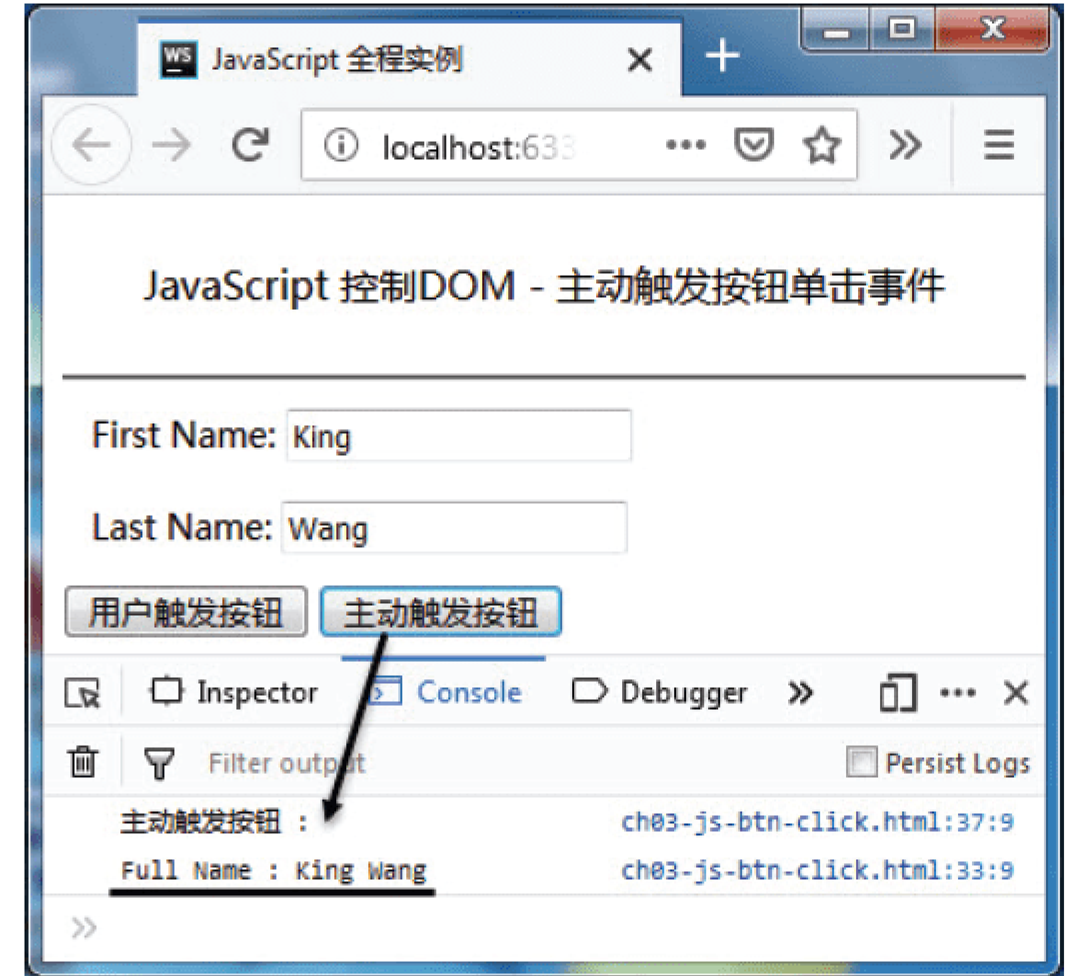


图 3.9　JavaScript 主动触发按钮单击事件

3.10　动态修改元素属性值

在前面的章节中介绍了如何遍历元素的全部属性，本节将在此基础上介绍如何动态修改元素的属性值。下面实现一个 JavaScript 动态修改元素属性值的代码实例。

【代码 3-10】（详见源代码目录 ch03-js-modify-attributes.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 控制 DOM - 动态修改元素属性值</nav>
</header>
<!-- 添加文档主体内容 -->
<p>Modify Attributes:
    <input type="text" class="css" name="name" id="id-name" value="King"/>
</p>
<input type="button" onclick="on_get_attr();" value="获取标签的全部属性"/>
<input type="button" onclick="on_modify_attr();"
        value="修改标签的全部属性"/>
</body>
```

```
19 <script type="text/javascript">
20     function on_modify_attr() {
21         var input = document.getElementsByTagName("input");
22         var attrs = input[0].attributes;
23         console.log("modify input's all attributes --- ");
24         var new_attrs = ["password", "newCSS", "pwd", "id-pwd", "123456"];
25         for (var i = 0; i < attrs.length; i++) {
26             attrs[i].value = new_attrs[i];
27         }
28         for(var j = 0; j < attrs.length; j++) {
29             console.log("attr name : " + attrs[j].name + ", value : " +
                            attrs[j].value);
30         }
31     }
32 </script>
33 </html>
```

关于【代码 3-10】的说明：

- 第 14 行代码通过标签<input>定义了一个文本框，并定义了 type="text"、class="css"、name="name"、id="id-name"和 value="King"这几个属性。
- 第 16 行和第 17 行代码分别定义了两个按钮控件：第一个按钮控件用于获取文本框（在第 14 行代码中定义）的全部原始属性，第二个按钮控件用于动态修改该文本框的全部属性。
- 第 20～31 行代码定义的函数实现了动态修改文本框全部属性的操作。其中，第 24 行代码定义的数组包括了全部新的属性值，比较有意思的是将文本框类型（type="text"）修改为了密码框类型（type="password"）。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 3.10 所示。

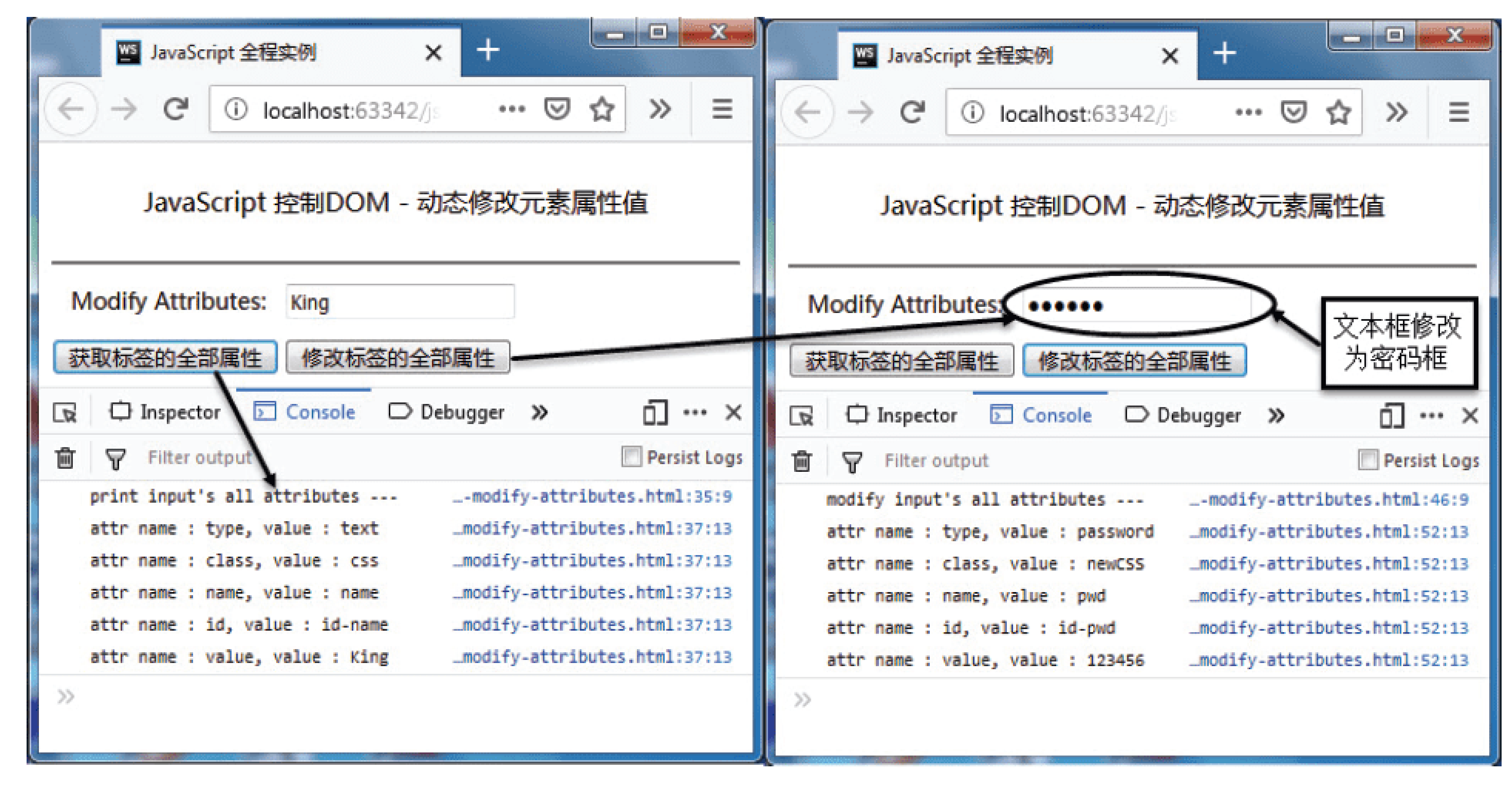


图 3.10　JavaScript 动态修改元素属性值

3.11 如何获取下拉列表的选项

一般来讲，获取下拉列表的选项主要是通过 selectIndex 属性来操作的，其实还有一个 index 索引属性也可以实现该操作。本节就分别介绍如何通过这两个属性来获取下拉列表的选项（包括获取索引和文本值）。

首先，介绍一个 JavaScript 通过 selectIndex 属性获取下拉列表选项的代码实例。

【代码 3-11】（详见源代码目录 ch03-js-select-selectIndex.html 文件）

```
<!doctype html>
<html lang="en">
<head>
  <!-- 添加文档头部内容 -->
  <title>JavaScript 全程实例</title>
</head>
<body>
  <!-- 添加文档主体内容 -->
  <header>
      <nav>JavaScript 控制 DOM - 获取下拉列表的选项(selectIndex)</nav>
  </header>
  <!-- 添加文档主体内容 -->
  <p>Lang:  
      <select id="id-select-lang">
          <option>CH</option>
          <option>EN</option>
          <option>FR</option>
          <option>GE</option>
          <option>RU</option>
          <option>HK</option>
      </select>
  </p>
  <input type="button" onclick="on_get_selIndex();"
          value="获取下拉列表选项索引" />
</body>
<script type="text/javascript">
    function on_get_selIndex() {
        var i = document.getElementById("id-select-lang").selectedIndex;
        var o = document.getElementById("id-select-lang").options[i];
        console.log("select index : " + i + "." + " select text : " + o.text);
    }
```

```
31 </script>
32 </html>
```

关于【代码 3-11】的说明：

- 第 14～21 行代码通过标签<select>定义了一个下拉菜单，并添加了一组选项。
- 第 26～30 行代码定义的函数实现了通过 selectIndex 属性操作下拉菜单选项的功能。其中，第 27 行代码通过 selectIndex 属性获取了下拉菜单选项的索引，第 28 行代码通过集合 options 对象获取了索引值对应的下拉菜单选项，第 29 行代码通过 text 属性在浏览器控制台中输出了下拉菜单选项的文本内容。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 3.11 所示。

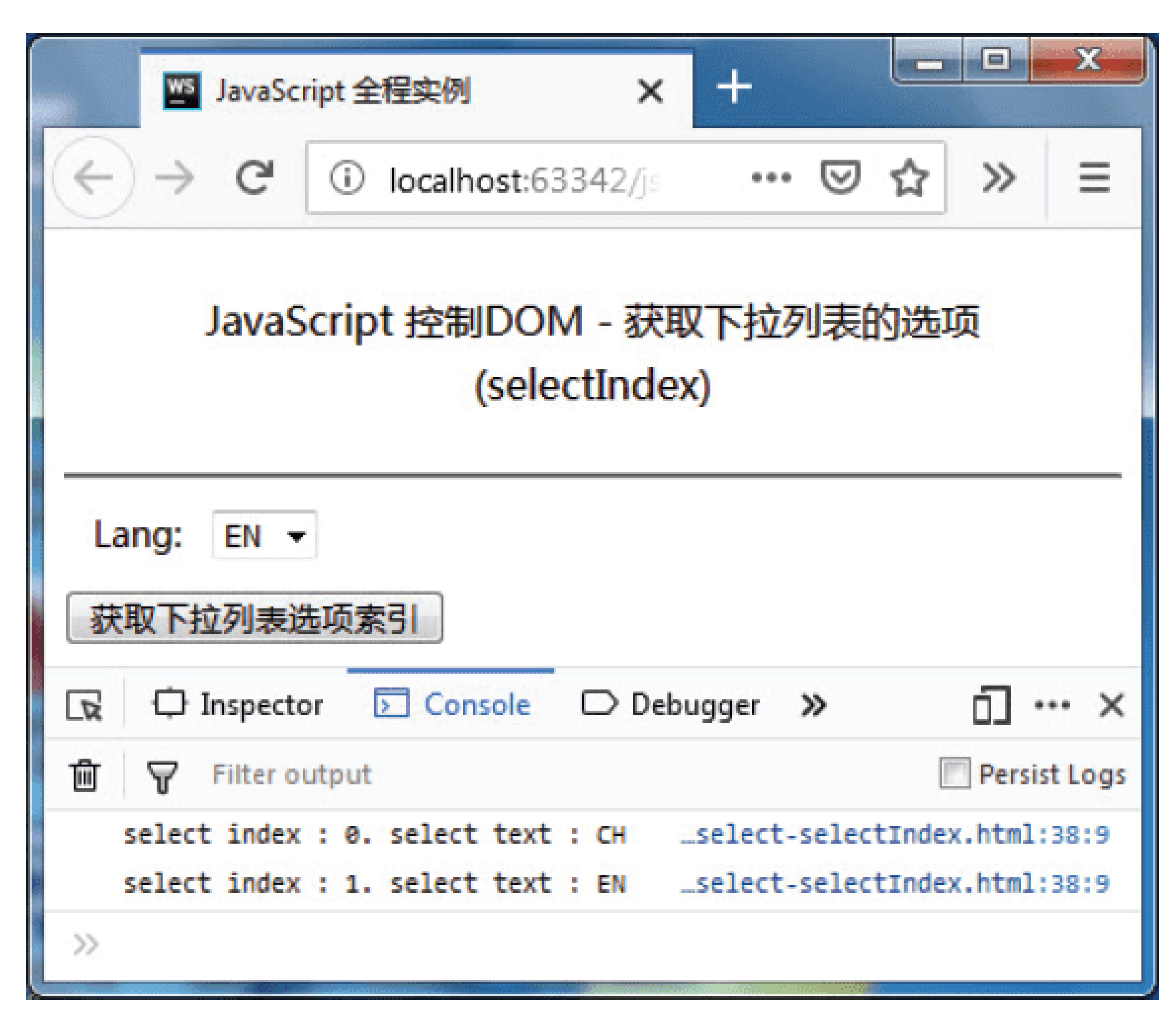


图 3.11 JavaScript 获取下拉列表的选项（selectIndex）

下面，再介绍一个 JavaScript 通过 index 属性获取下拉列表选项的代码实例。

【代码 3-12】（详见源代码目录 ch03-js-select-index.html 文件）

```
01 <script type="text/javascript">
02     function on_get_sel_index() {
03         var i = document.getElementById("id-select-lang").selectedIndex;
04         var options = document.getElementsByTagName("option");
05         console.log("select index : " + options[i].index + "." +
                       " select text : " + options[i].text);
06     }
07 </script>
```

关于【代码 3-12】的说明：

- 本代码是在【代码 3-11】的基础上修改而成的，主要区别就是在第 05 行代码中使用 index 属性获取了下拉菜单选项的索引值。

读者可以自行测试一下该代码实例，与【代码 3-11】的效果是完全一致的。

3.12　实现电话拨号键盘

前面简单介绍了如何使用下拉菜单选项的 index 属性，其实 index 属性还可以用于控制同级的多个同类控件，比如常见的按钮控件。对于同级的多个按钮控件，可以用 index 属性作为按钮的索引，以区分多个同级按钮。

下面给出一个通过使用多个同级按钮控件并借助 index 属性实现电话拨号键盘的 JavaScript 代码实例。

【代码 3-13】（详见源代码目录 ch03-js-btn-telphone.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 控制 DOM - 实现电话拨号键盘</nav>
</header>
<!-- 添加文档主体内容 -->
<div class="css-div-button">
    <button>1</button>
    <button>2</button>
    <button>3</button><br>
    <button>4</button>
    <button>5</button>
    <button>6</button><br>
    <button>7</button>
    <button>8</button>
    <button>9</button><br>
    <button>*</button>
    <button>0</button>
    <button>#</button><br>
    <input type="text" id="id-input-tel" readonly />
</div>
</body>
<script type="text/javascript">
```

```
30     window.onload = function () {
31         var arrTel = ["1", "2", "3", "4", "5", "6", "7", "8", "9", "*",
                     "0", "#"];
32         var arrBtn = document.getElementsByTagName('button');
33         var tel = document.getElementById("id-input-tel");
34         for (var i = 0; i < arrBtn.length; i++) {
35             arrBtn[i].index = i;
36             arrBtn[i].onclick = function () {
37                 tel.value += arrTel[this.index];
38             };
39         }
40     };
41 </script>
42 </html>
```

关于【代码 3-13】的说明：

- 第 14 ~ 25 行代码通过一组<button>按钮控件实现了一个电话拨号键盘(包括 0 ~ 9 数字键、“*”键和 “#” 键)。
- 第 26 行代码定义了一个文本框（只读），用于显示电话号码。
- 第 31 行代码定义了一个数组，包含了电话拨号键盘的数字和字符。
- 第 32 行代码获取了全部<burron>按钮控件的对象集合（arrBtn）。
- 第 34 ~ 39 行代码通过 for 循环语句遍历了全部按钮控件的对象集合（arrBtn）。其中，第 35 行代码为每一个按钮对象的 index 属性定义了索引值 (自变量 i); 第 36 ~ 38 行代码为每一个按钮对象定义了 onclick 事件处理方法，用于将用户的拨号操作显示在第 26 行代码定义的只读文本框中。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 3.12 所示。

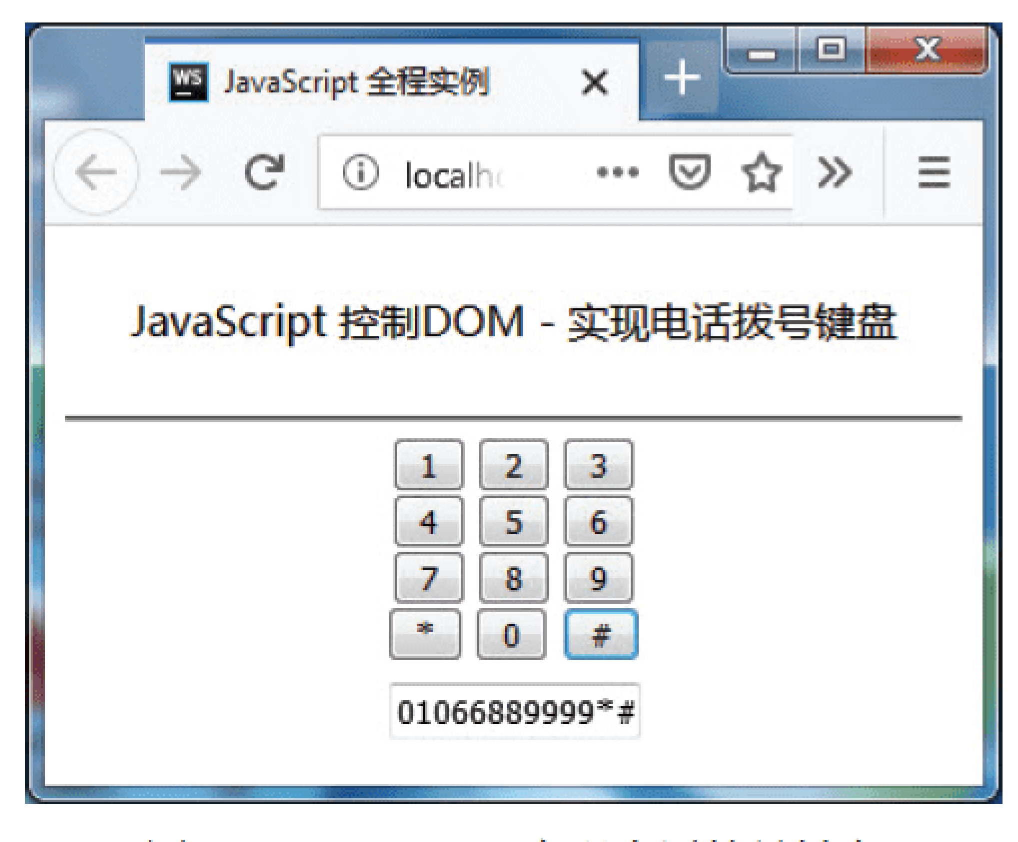


图 3.12　JavaScript 实现电话拨号键盘

第 4 章　按 钮 特 效

本章介绍如何通过 JavaScript 来实现各种按钮特效，通过这些特效来丰富 HTML 文档的设计手段和页面效果。

4.1　按钮概述

HTML 语法规范中定义，可以使用<input type="button|submit|reset">标签来定义按钮（按钮|提交|重置），或者直接使用<button>标签可以定义按钮，目前几乎所有的主流浏览器均支持这两种定义方式。

在具体应用时，这两种定义方式还是略有区别的。<input type="button|submit|reset">标签的浏览器兼容性比<button>标签更好一些，<button>标签在 IE 浏览器和非 IE 浏览器中会有所区别（当然现在绝大部分用户都没在使用 IE 浏览器了）。

当然任何事情都是有利有弊的，<button>标签相对于<input type="button|submit|reset">标签提供了更为强大的功能和更丰富的内容。在<button></button>标签之间，设计人员可以放置文本、图像和多媒体等内容，这点是<input type="button|submit|reset">标签所不具备的功能。

HTML 语法规范为按钮对象定义了一组事件方法(比如:单击事件和焦点事件)，通过 JavaScript 脚本语言可以对这些事件进行编程控制，实现更为强大的页面功能。同时，通过 JavaScript 脚本语言还可以控制按钮的 CSS 样式代码，进而展现出更为丰富的页面特效。

4.2　为按钮添加背景颜色

在 HTML DOM 对象中，定义有一个 Style 对象，用于表示风格样式，绝大多数的页面元素均可以通过该对象来定义风格样式。在 Style 对象中，定义了一个 backgroundColor 属性，表示元素的背景颜色。如果将 backgroundColor 属性应用于按钮<button>元素，就可以为按钮控件添加各种背景颜色了。

下面简单介绍一个 JavaScript 通过 backgroundColor 属性为按钮控件添加背景颜色的代码实例。

【代码 4-1】（详见源代码目录 ch04-js-btn-backgroundColor.html 文件）

```
<!doctype html>
<html lang="en">
<head>
  <!-- 添加文档头部内容 -->
```

```
05   <title>JavaScript 全程实例</title>
06 </head>
07 <body>
08   <!-- 添加文档主体内容 -->
09   <header>
10       <nav>JavaScript 按钮特效 - 为按钮添加背景颜色</nav>
11   </header>
12   <!-- 添加文档主体内容 -->
13   <div class="css-div-button">
14       <button>Red</button>
15       <button>Yellow</button>
16       <button>Blue</button>
17   </div>
18 </body>
19 <script type="text/javascript">
20   var colorBtn = ["red", "yellow", "blue"];
21   var arrBtn = document.getElementsByTagName("button");
22   for(var i=0; i<arrBtn.length; i++) {
23      arrBtn[i].style.backgroundColor = colorBtn[i];
24   }
25 </script>
26 </html>
```

关于【代码 4-1】的说明：

- 第 14 ~ 16 行代码通过<button>标签定义一组按钮，后面将为这组按钮定义不同的背景颜色。
- 第 20 行代码定义了一个数组（colorBtn），分别定义了 3 种颜色（"red"，"yellow"和"blue"）。
- 第 21 行代码获取了按钮对象集合（arrBtn）。
- 第 22 ~ 24 行代码通过一个 for 语句依次为按钮对象（arrBtn[i]）定义背景颜色（colorBtn[i]），具体就是通过 Style 对象的 backgroundColor 属性来操作的。

由于书籍印刷的原因，这里就不给出页面效果图了。按钮控件的背景颜色效果，读者可以自行运行源代码测试查看。

4.3 不同按钮提交到不同的表单地址

对于 HTML DOM 对象中的表单<form>元素而言，一个表单只能提交到一个由 action 属性定义的服务器端页面上，因为 action 属性就只能定义一个 url 链接地址。那么能不能通过 JavaScript 实现一个表单提交到不同服务器端地址页面上去呢？答案是肯定的，通过在表单中定义多个按钮就可以实现提交到不同表单地址的功能。

下面就介绍一个通过 JavaScript 实现不同按钮提交到不同表单地址的代码实例。

【代码 4-2】（详见源代码目录 ch04-js-btn-multiForm.html 文件）

```
01  <!doctype html>
02  <html lang="en">
03  <head>
04      <!-- 添加文档头部内容 -->
05      <title>JavaScript 全程实例</title>
06  </head>
07  <body>
08  <!-- 添加文档主体内容 -->
09  <header>
10      <nav>不同按钮提交到不同表单地址</nav>
11  </header>
12  <!-- 添加文档主体内容 -->
13  <form name="formMulti" method="get" enctype="multipart/form-data"
          action="">
14      <p>First Name:  <input type="text" name="fname"
                 value="King"/></p>
15      <p>Last Name:  <input type="text" name="lname"
                 value="Wang"/></p>
16      <button type="submit" name="btnCh" onclick="submit_ch();">
                提交(CH)</button>
17      <button type="submit" name="btnEn" onclick="submit_en();">
                提交(EN)</button>
18  </form>
19  </body>
20  <script language="javascript">
21      function submit_ch() {
22          document.formMulti.action = "multiForm-ch.php";
23          document.formMulti.submit();
24      }
25      function submit_en() {
26          document.formMulti.action = "multiForm-en.php";
27          document.formMulti.submit();
28      }
29  </script>
30  </html>
```

关于【代码 4-2】的说明：

- 第 13 ~ 18 行代码通过<form>标签定义一个表单，其中的 action 属性没有定义具体属性值。
- 第 16 ~ 17 行代码通过<button type="submit">标签定义了两个按钮控件，并分别定义了不同的 onclick 事件处理方法（submit_ch()、submit_en()），用于实现提交到不同的表单地址的操作。

- 第 21～24 行代码是 submit_ch()方法的实现过程，第 22 行代码通过 action 属性定义了表单提交的服务器端 PHP 页面（multiForm-ch.php），第 23 行代码通过调用 submit()方法实现提交表单的操作。
- 同样的，第 25～28 行代码是 submit_en()方法的实现过程，以及表单提交的服务器端 PHP 页面（multiForm-ch.php）。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 4.1 和图 4.2 所示。

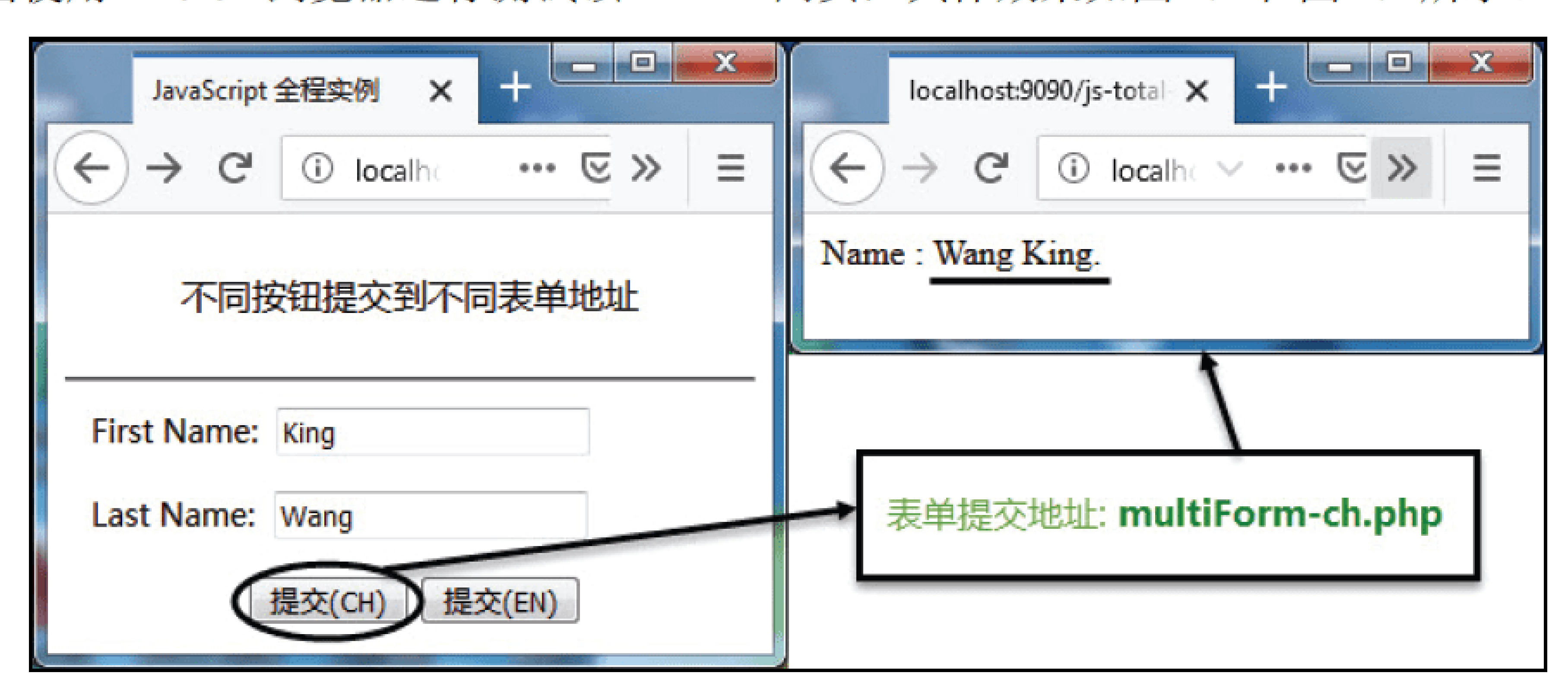


图 4.1　JavaScript 不同按钮提交到不同的表单地址（一）

如图 4.1 所示，第一个提交按钮将表单提交到服务器端 PHP 页面（multiForm-ch.php），页面中显示出了中文习惯的名字。

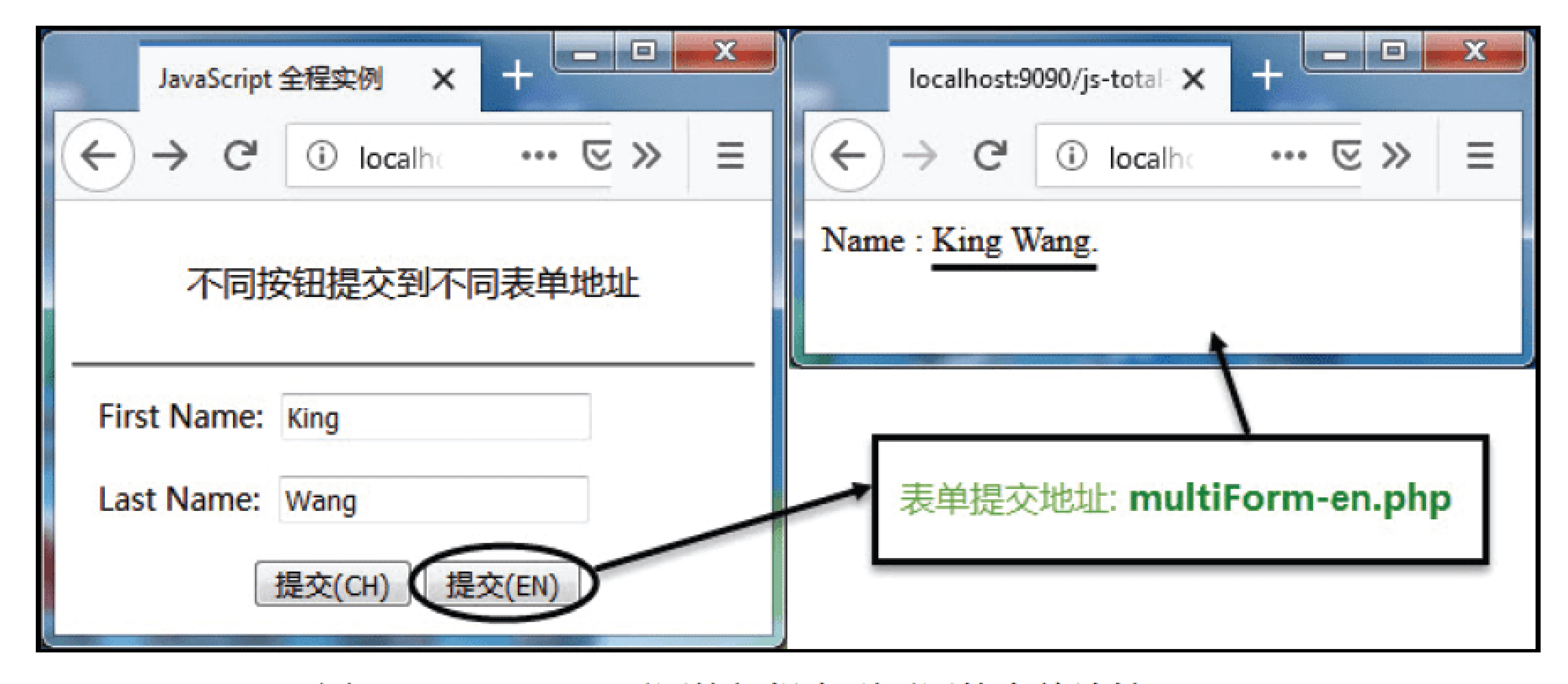


图 4.2　JavaScript 不同按钮提交到不同的表单地址（二）

如图 4.2 所示，第二个提交按钮将表单提交到服务器端 PHP 页面（multiForm-en.php），页面中显示出了英文习惯的名字。

4.4　避免回车键自动提交表单

一般情况下，如果在一个表单中只定义了一个 submit 类型的提交按钮，那么当用户在表单输入完后按下回车键（Enter）就可以实现表单的提交操作。不过在有些情况下，这种看似便捷的功

能也会带来不必要的烦恼，那就是如果用户进行了误操作（在还没有输入完表单的情况下，就误按回车键提交了表单）怎么办？这个时候就需要通过 JavaScript 来禁用回车键（Enter）功能了。

【代码 4-3】（详见源代码目录 ch04-js-form-no-enter.html 文件）

```
01 <!doctype html>
02 <html lang="en">
03 <head>
04     <!-- 添加文档头部内容 -->
05     <title>JavaScript 全程实例</title>
06 </head>
07 <body>
08 <!-- 添加文档主体内容 -->
09 <header>
10     <nav>避免回车键自动提交表单</nav>
11 </header>
12 <!-- 添加文档主体内容 -->
13 <form name="formNoEnter" method="get" action="form-no-enter.php">
14     <p>First Name: <input type="text" name="fname" id="id-fname"
                             value="King"/></p>
15     <p>Last Name: <input type="text" name="lname" id="id-lname"
                            value="Wang"/></p>
16     <button type="submit" id="id-submit">提交表单</button>
17 </form>
18 </body>
19 <script type="text/javascript">
20     window.onload = function () {
21         document.formNoEnter.onkeypress = function (ev) {
22             var ev = window.event || ev;
23             if (ev.keyCode == 13 || ev.which == 13) {
24                 console.log("Info : press enter no submit.");
25                 return false;
26             }
27         }
28     }
29 </script>
30 </html>
```

关于【代码 4-3】的说明：

- 第 13 ~ 17 行代码通过<form>标签定义一个表单（formNoEnter），同时定义了 action 属性（form-no-enter.php）。其中，第 16 行代码通过<button type="submit">标签定义了一个提交按钮；表单中如果定义了该 submit 类型的按钮，如果不对回车键（Enter）进行屏蔽处理的话，那么当用户按下回车键后，默认就会执行提交表单的操作（读者可以自行测试一下）。

- 第 21～27 行代码为表单（formNoEnter）定义了按下按键 onkeypress 事件处理方法。其中，第 23 行代码通过判断回车键（Enter）按键 ASCII 编码（13）来屏蔽用户操作，这里要注意浏览器兼容性代码的写法；如果判断结果为“真”，就在浏览器控制台中输出一行提示信息（第 24 行代码），并通过返回 false 来屏蔽按键操作（第 25 行代码）。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 4.3 所示。

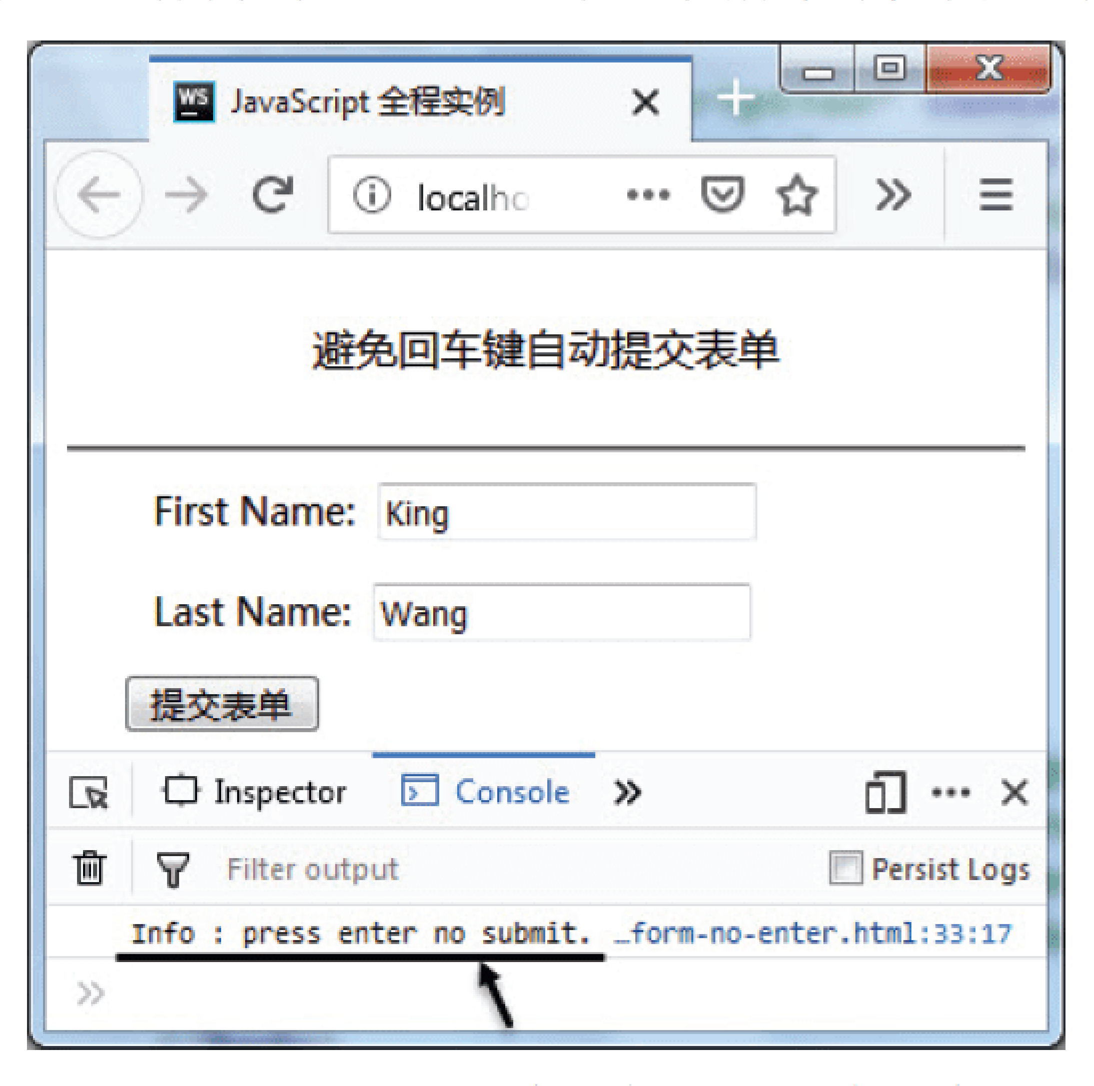


图 4.3　JavaScript 避免回车键自动提交表单

如图 4.3 中箭头所示，当用户按下回车键（Enter）后，表单提交操作没有生效，而在浏览器控制台中显示了第 24 行代码定义的提示信息。

4.5　按钮在单击后自动失效

在某些 HTML 页面应用中，可能需要实现按钮在单击一次后就自动失效的效果，通过设置按钮控件的 disabled 属性就可以实现该功能。下面给出一个通过 JavaScript 实现按钮在单击后自动失效的代码实例。

【代码 4-4】（详见源代码目录 ch04-js-btn-click-once.html 文件）

```
<!doctype html>
<html lang="en">
<head>
  <!-- 添加文档头部内容 -->
  <title>JavaScript 全程实例</title>
</head>
<body>
  <!-- 添加文档主体内容 -->
  <header>
```

```
10        <nav>按钮在单击后自动失效</nav>
11    </header>
12    <!-- 添加文档主体内容 -->
13    <div>测试按钮:
14        <button id="id-click-once">Click Once</button>
15    </div>
16    <div>重新激活:
17        <button id="id-re-active">Re-Active</button>
18    </div>
19 </body>
20 <script type="text/javascript">
21     document.getElementById("id-click-once").onclick = function (e) {
22         e.target.disabled = true;
23     };
24     document.getElementById("id-re-active").onclick = function (e) {
25         document.getElementById("id-click-once").disabled = false;
26     };
27 </script>
28 </html>
```

关于【代码 4-4】的说明：

- 第 14 行和第 17 行代码通过<button>标签分别定义了两个按钮，第一个按钮（id="id-click-once"）用于测试在单击后自动失效的功能，第二个按钮（id="id-re-active"）用于重新激活第一个按钮。
- 第 21～23 行代码为第一个按钮定义了 onclick 事件处理方法，其中第 22 行代码通过设定 disabled 属性（=true）让该按钮失效。
- 第 24～26 行代码为第二个按钮定义了 onclick 事件处理方法，其中第 25 行代码通过设定 disabled 属性（=false）让第一个按钮重新被激活。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 4.4 所示。

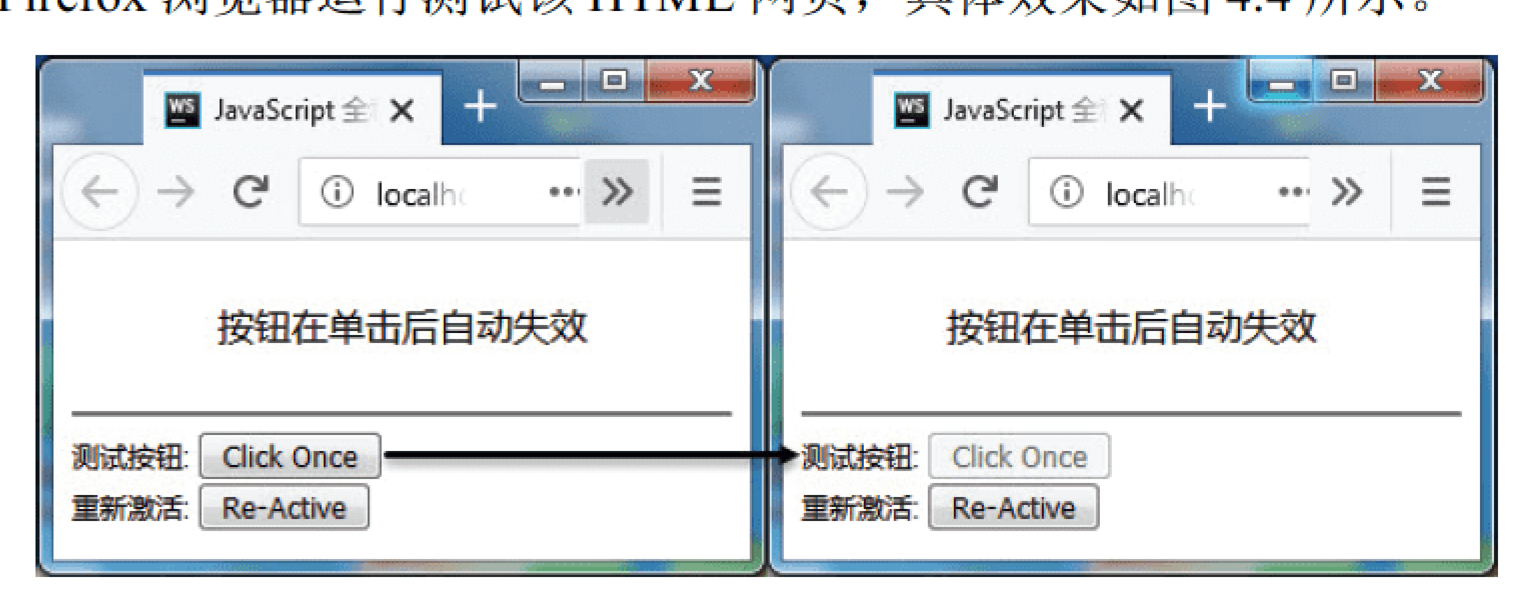


图 4.4　JavaScript 控制按钮在单击后自动失效

如图 4.4 中箭头所示，当用户按下第一个按钮（Click Once）后，该按钮就自动失效了。用户可以通过第二个按钮（Re-Active）再次激活第一个按钮。

4.6 为删除功能按钮添加确认提醒

一般情况下，删除操作都是一种风险比较大的行为，所以设计人员一般都会为删除功能添加上一个再次确认提醒的功能。在 HTML DOM 对象中定义有一个 Window（窗体）对象，该对象中定义了一个 confirm()方法，用于显示一个带有指定消息并带有 OK 按钮及取消按钮的对话框（一般称为确认对话框）。

下面介绍一个 JavaScript 通过 confirm()方法为删除按钮添加确认提醒功能的代码实例。

【代码 4-5】（详见源代码目录 ch04-js-btn-confirm.html 文件）

```
01 <!doctype html>
02 <html lang="en">
03 <head>
04   <!-- 添加文档头部内容 -->
05   <title>JavaScript 全程实例</title>
06 </head>
07 <body>
08   <!-- 添加文档主体内容 -->
09   <header>
10       <nav>为删除功能按钮添加确认提醒</nav>
11   </header>
12   <!-- 添加文档主体内容 -->
13   <button id="id-btn-confirm">删除按钮</button>
14   <p>删除按钮 --- 删除内容</p>
15 </body>
16 <script type="text/javascript">
17    document.getElementById("id-btn-confirm").onclick = function (e) {
18        if(confirm("Please confirm to del ?")) {
19            document.getElementsByTagName("p")[0].innerText = "";
20        }
21    };
22 </script>
23 </html>
```

关于【代码 4-5】的说明：

- 第 13 行代码通过<button id="id-btn-confirm">标签定义一个按钮，用于删除第 14 行代码定义的<p>标签中的内容。
- 第 17 ~ 21 行代码为<button id="id-btn-confirm">标签定义了 onclick 单击事件处理方法。其中，第 18 ~ 20 行代码通过判断调用 confirm()方法的返回值让用户选择是否执行删除操作。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 4.5 所示。

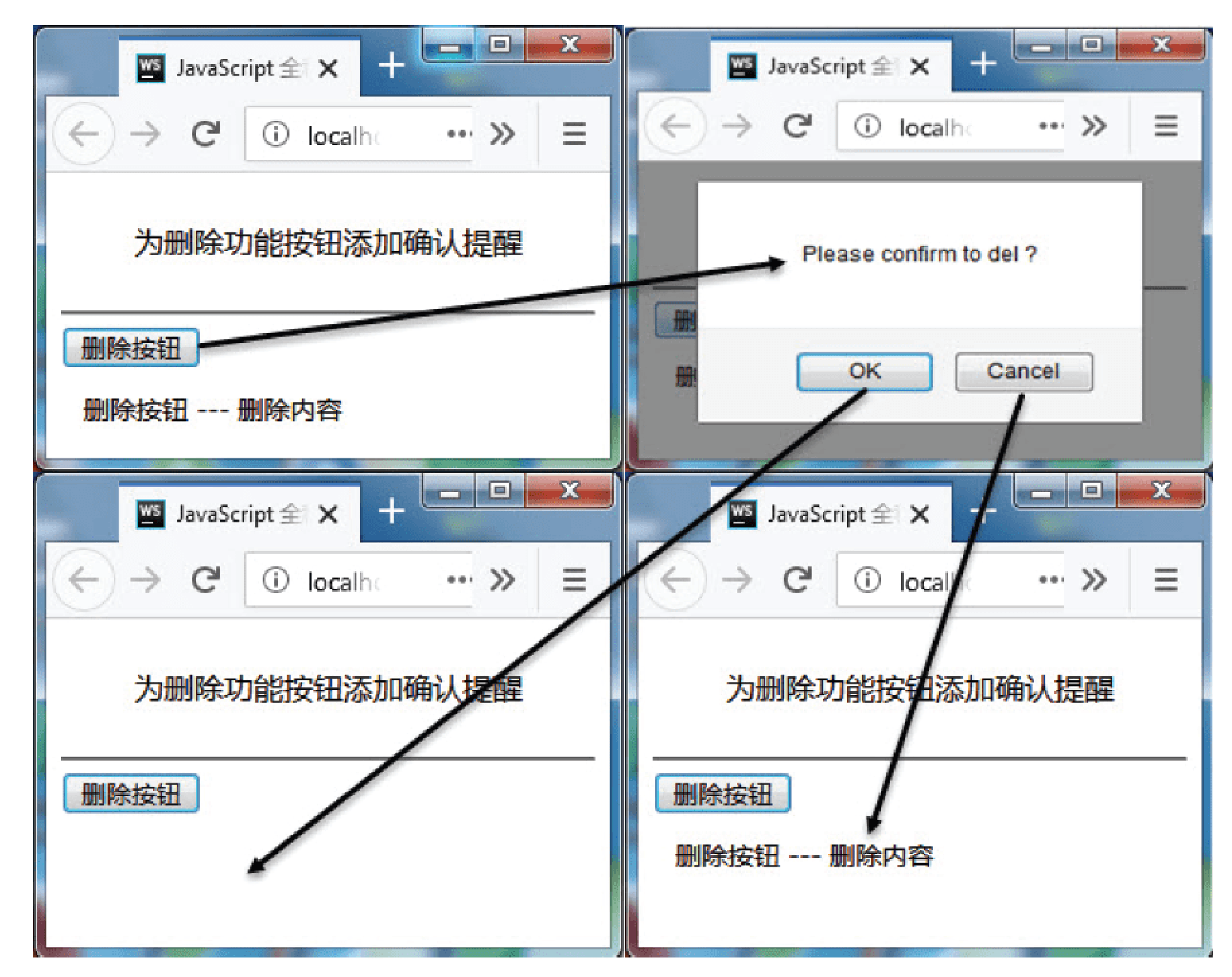


图 4.5　JavaScript 为删除功能按钮添加确认提醒

如图 4.5 中箭头所示，当用户按下“删除按钮”后，页面中会先弹出一个确认框，然后根据用户的选择确认是否删除内容。

4.7　根据状态展示不同样式按钮

在某些 HTML 页面应用中，设计人员需要根据特定状态将页面中的按钮展示出不同的风格，此时就需要通过 JavaScript 来动态改变按钮的样式。下面给出一个通过 JavaScript 实现根据状态展示不同样式按钮的代码实例。

【代码 4-6】（详见源代码目录 ch04-js-btn-status.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
```

```
<!-- 添加文档主体内容 -->
<header>
    <nav>根据状态展示不同样式按钮</nav>
</header>
<!-- 添加文档主体内容 -->
<div css="css-btn-status">
    <button id="id-btn-status" style="width:128px;height:32px">正常状态</button>
    <p>当前的状态：</p>
</div>
<button id="id-btn-normal" onclick="on_normal_status()">正常状态</button>
<button id="id-btn-running" onclick="on_running_status()">运行状态</button>
<button id="id-btn-stop" onclick="on_stop_status()">停止状态</button>
<button id="id-btn-disable" onclick="on_disable_status()">禁用状态</button>
</body>
<script type="text/javascript">
    var curStatus = "";
    function on_normal_status() {
        var v_btn = document.getElementById('id-btn-status');
        v_btn.disabled = false;        //恢复可用状态
        v_btn.innerText = "正常状态";
        v_btn.style.color = "";
        v_btn.style.fontWeight = "";
        v_btn.style.backgroundColor = "";
        curStatus = '正常状态';           //设置正常状态显示值
        var p = document.getElementsByTagName("p")[0];
        p.innerHTML = '当前的状态：' + curStatus;
    }
    function on_running_status() {
        var v_btn = document.getElementById('id-btn-status');
        v_btn.disabled = false;        //恢复可用状态
        v_btn.innerText = "运行状态";
        v_btn.style.color = "#fff";
        v_btn.style.fontWeight = "bold";
        v_btn.style.backgroundColor = "#666";
        curStatus = '运行状态';           //设置运行状态显示值
        var p = document.getElementsByTagName("p")[0];
        p.innerHTML = '当前的状态：' + curStatus;
    }
    function on_stop_status() {
        var v_btn = document.getElementById('id-btn-status');
```

```
48          v_btn.disabled = false;        //恢复可用状态
49          v_btn.innerText = "停止状态";
50          v_btn.style.color = "#888";
51          v_btn.style.fontWeight = "bold";
52          v_btn.style.backgroundColor = "#ccc";
53          curStatus = '停止状态';            //设置停止状态显示值
54          var p = document.getElementsByTagName("p")[0];
55          p.innerHTML = '当前的状态：' + curStatus;
56       }
57       function on_disable_status() {
58          var v_btn = document.getElementById('id-btn-status');
59          v_btn.disabled = true;        //设置不可用为 true
60          v_btn.innerText = "禁用状态";
61          v_btn.style.color = "#ccc";
62          v_btn.style.fontWeight = "normal";
63          v_btn.style.backgroundColor = "#eee";
64          curStatus = '禁用状态';        //设置不可用状态显示值
65          var p = document.getElementsByTagName("p")[0];
66          p.innerHTML = '当前的状态：' + curStatus;//显示当前状态
67       }
68   </script>
69   </html>
```

关于【代码 4-6】的说明：

- 第 14 行代码通过<button id="id-btn-status">标签定义一个按钮，用于演示按钮的不同状态。
- 第 15 行代码定义的<p>标签用于显示按钮当前的状态。
- 第 17～20 行代码通过<button>标签分别定义了 4 个按钮，用于切换第 14 行代码中定义的<button id="id-btn-status">按钮的状态。
- 第 24～34 行、第 35～45 行、第 46～56 行、第 57～67 行代码分别定义了第 17～20 行代码中 4 个按钮的 onclick 事件处理方法。其中，使用 disabled 属性定义按钮是否被禁用，使用 innerText 属性定义按钮不同状态下的显示内容，使用 color、fontWeight 和 backgroundColor 属性分别定义按钮不同状态下的样式。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 4.6 所示。

如图 4.6 中所示，4 个页面中分别展示了按钮的 4 种状态（正常状态、运行状态、停止状态和禁用状态）。

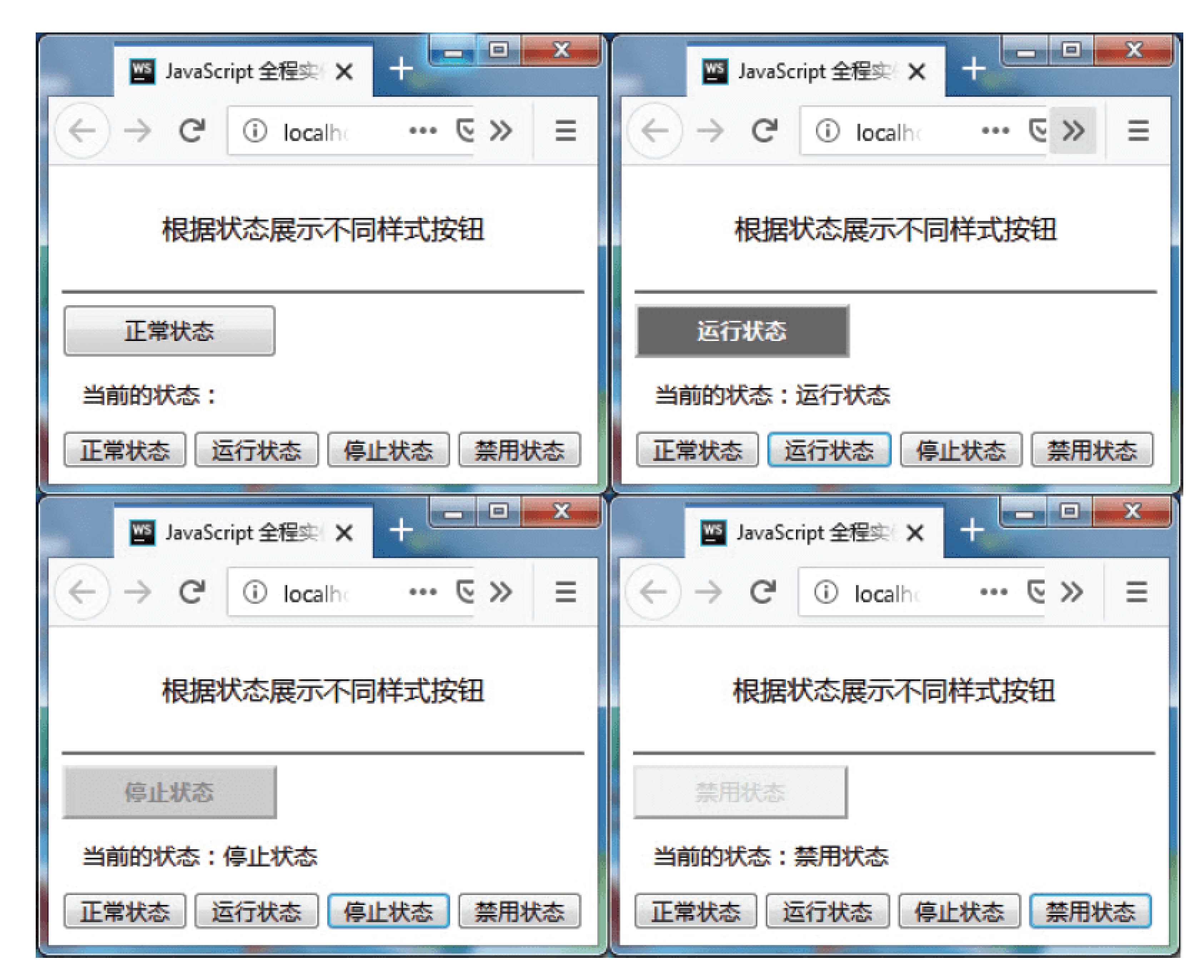


图 4.6　JavaScript 根据状态展示不同样式按钮

4.8　注册按钮倒计时效果

相信大多数读者都遇到过下面的场景，用户注册页面中注册按钮是带有倒计时功能的，这样用户就必须要等待倒计时完成后才能操作（好意提醒一下用户再次核对个人信息是否准确）。那么，这个功能是如何实现的呢？还是利用 HTML DOM 对象中定义的 Window（窗体）对象。该对象中定义了一个 setInterval()方法，可以按照指定的周期（以毫秒计）来调用函数或计算表达式。注意，这个 setInterval()方法会不停地循环调用函数，直到使用 clearInterval()方法清除计时器或者当前窗口被关闭后才会停止。

下面就介绍一个 JavaScript 通过 setInterval()方法实现具有倒计时效果的注册按钮的代码实例。

【代码 4-7】（详见源代码目录 ch04-js-btn-reg-timer.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
```

```
10     <nav>注册按钮倒计时效果</nav>
11  </header>
12  <!-- 添加文档主体内容 -->
13  <div css="css-btn-timer">
14     <p>
15     <button id="id-btn-reg-timer" style="width:150px;
                                     height:32px">注册 (剩余 10 秒) </button>
16     </p>
17  </div>
18  </body>
19  <script type="text/javascript">
20     var btnRegTimer = null;  //定时器
21     window.onload = function () {
22        var v_btn = document.getElementById("id-btn-reg-timer");
23        v_btn.disabled = true;
24        v_btn.innerText = "注册（剩余 10 秒）";
25        window.clearInterval(btnRegTimer);   //取消定时器
26        var btnRegSec = 10;                  //开始 10 秒倒计时
27        btnRegTimer = window.setInterval(function () {
28           if (btnRegSec == 0) {
29              v_btn.disabled = false;
30              v_btn.innerText = "注册";
31              window.clearInterval(btnRegTimer);   //取消定时器
32           } else {
33              btnRegSec--;                         //让倒计时秒数自减
34              v_btn.innerText = "注册（剩余 " + btnRegSec + " 秒）";
35           }
36        }, 1000);
37     };
38  </script>
39  </html>
```

关于【代码 4-7】的说明：

- 第 15 行代码通过<button id="id-btn-reg-timer">标签定义一个按钮，用于演示注册按钮具有倒计时效果。
- 第 20 行代码定义并初始化了一个空的计时器（btnRegTimer = null）。
- 第 21 ~ 37 行代码定义了 Window 对象的 onload 事件处理方法。其中，第 26 行代码定义了一个用于累计倒计时时间变化的变量（btnRegSec），并初始化为 10 秒；第 27 ~ 36 行代码通过 setInterval()方法对计时器（btnRegTimer）进行了定义，通过 disabled 属性来定义注册按钮（<button id="id-btn-reg-timer">）的样式，通过 innerText 属性定义注册按钮的倒计时文本内容（倒计时秒数的变化）。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 4.7 所示。

图 4.7　JavaScript 注册按钮倒计时效果

如图 4.7 中箭头所示，当倒计时完成后，页面中按钮的禁用状态被解除，并变为可用状态。

4.9　计时器按钮

前面介绍了具有倒计时效果的注册按钮，其实还有一种类似秒表计时器功能的按钮，同样是通过 JavaScript 计时器来实现的。一般来讲，秒表计时器具有开始计时、暂停计时、恢复计时和停止计时这几个基本功能且时间单位显示到毫秒，就如同百米竞赛中裁判员使用的秒表一样。

下面就具体介绍一个 JavaScript 通过 setInterval()方法实现具有计时器按钮功能的代码实例。

【代码 4-8】（详见源代码目录 ch04-js-btn-timer.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <title>JavaScript 全程实例</title>
</head>
<body onload="init();">
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 按钮特效 - 计时器按钮</nav>
</header>
<!-- 添加文档主体内容 -->
```

```
<div css="css-btn-timer">
    <p>
        <button id="id-btn-timer" disabled>计时器按钮(0)秒</button>
    </p>
    <p>
        <button id="id-btn-start" onclick="on_start_timer()">开始倒计时
                                          </button>
        <button id="id-btn-pause" onclick="on_pause_timer()">暂停倒计时
                                          </button>
        <button id="id-btn-restore" onclick="on_restore_timer()">恢复倒计时
                                          </button>
        <button id="id-btn-stop" onclick="on_stop_timer()">停止倒计时
                                          </button>
    </p>
</div>
</body>
<script type="text/javascript">
    var myTimer = null;       // TODO: 定义计时器
    var mySec = 0;            // TODO: 定义累加器
    function init() {
        var v_btn = document.getElementById('id-btn-timer');
        v_btn.innerText = "计时器按钮(0)秒";
        var v_btn_pause = document.getElementById('id-btn-pause');
        v_btn_pause.disabled = true;
        var v_btn_restore = document.getElementById('id-btn-restore');
        v_btn_restore.disabled = true;
        var v_btn_stop = document.getElementById('id-btn-stop');
        v_btn_stop.disabled = true;
    }
    function on_start_timer() {
        var v_btn = document.getElementById('id-btn-timer');
        var v_btn_start = document.getElementById('id-btn-start');
        v_btn_start.disabled = true;
        var v_btn_pause = document.getElementById('id-btn-pause');
        v_btn_pause.disabled = false;
        var v_btn_stop = document.getElementById('id-btn-stop');
        v_btn_stop.disabled = false;
        window.clearInterval(myTimer);     // TODO: 清除计时器
        mySec = 0;                         // TODO: 计时器清零
        myTimer = window.setInterval(function () {
            mySec++;                       // TODO: 计时器累加
            v_btn.innerText = "计时器按钮(" + mySec/1000 + ")秒";
        }, 1);
```

```
51      }
52      function on_pause_timer() {
53          var v_btn = document.getElementById('id-btn-timer');
54          var v_btn_pause = document.getElementById('id-btn-pause');
55          v_btn_pause.disabled = true;
56          var v_btn_restore = document.getElementById('id-btn-restore');
57          v_btn_restore.disabled = false;
58          window.clearInterval(myTimer);        // TODO: 清除计时器
59      }
60      function on_restore_timer() {
61          var v_btn = document.getElementById('id-btn-timer');
62          var v_btn_restore = document.getElementById('id-btn-restore');
63          v_btn_restore.disabled = true;
64          var v_btn_pause = document.getElementById('id-btn-pause');
65          v_btn_pause.disabled = false;
66          myTimer = window.setInterval(function () {
67              mySec++;                              //TODO: 计时器累加
68              v_btn.innerText = "计时器按钮(" + mySec/1000 + ")秒";
69          }, 1);
70      }
71      function on_stop_timer() {
72          var v_btn_stop = document.getElementById('id-btn-stop');
73          v_btn_stop.disabled = true;
74          var v_btn_start = document.getElementById('id-btn-start');
75          v_btn_start.disabled = false;
76          var v_btn_pause = document.getElementById('id-btn-pause');
77          v_btn_pause.disabled = true;
78          var v_btn_restore = document.getElementById('id-btn-restore');
79          v_btn_restore.disabled = true;
80          window.clearInterval(myTimer);            // TODO: 清除计时器
81          mySec = 0;                                // TODO: 计时器清零
82          var v_btn = document.getElementById('id-btn-timer');
83          v_btn.innerText = "计时器按钮(0)秒";
84      }
85  </script>
86  </html>
```

关于【代码 4-8】的说明：

- 第 14 行代码通过<button id="id-btn-timer">标签定义一个按钮，用于演示计时器按钮的效果。
- 第 17 ~ 20 行代码通过<button>标签定义一组按钮，分别用于实现开始倒计时、暂停倒计时、恢复倒计时和停止倒计时这几个基本功能。
- 第 25 行代码定义并初始化了一个空的计时器（myTimer = null），第 26 行代码定义并初始化

了一个累加器（mySec＝0）。

- 第 27～36 行代码定义了 Window 对象的 onload 事件处理方法（init()），分别定义了开始倒计时、暂停倒计时、恢复倒计时和停止倒计时这 4 个按钮的状态，主要是通过 disabled 属性来实现启动或禁用按钮的功能。
- 第 37～51 行、第 52～59 行、第 60～70 行和第 71～84 行代码定义的函数，分别实现了开始倒计时、暂停倒计时、恢复倒计时和停止倒计时的功能，主要是通过 setInterval()方法和 clearInterval()定义和清除计时器（myTimer），同时通过累加器（mySec）实现计时功能。同时，还通过按钮的 disabled 属性实现了启用按钮和禁用按钮的功能切换逻辑。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 4.8 所示。在 4 张页面截图中分别演示了初始状态、开始倒计时、暂停倒计时和恢复倒计时的状态，而单击“停止倒计时”按钮则会恢复到初始状态。

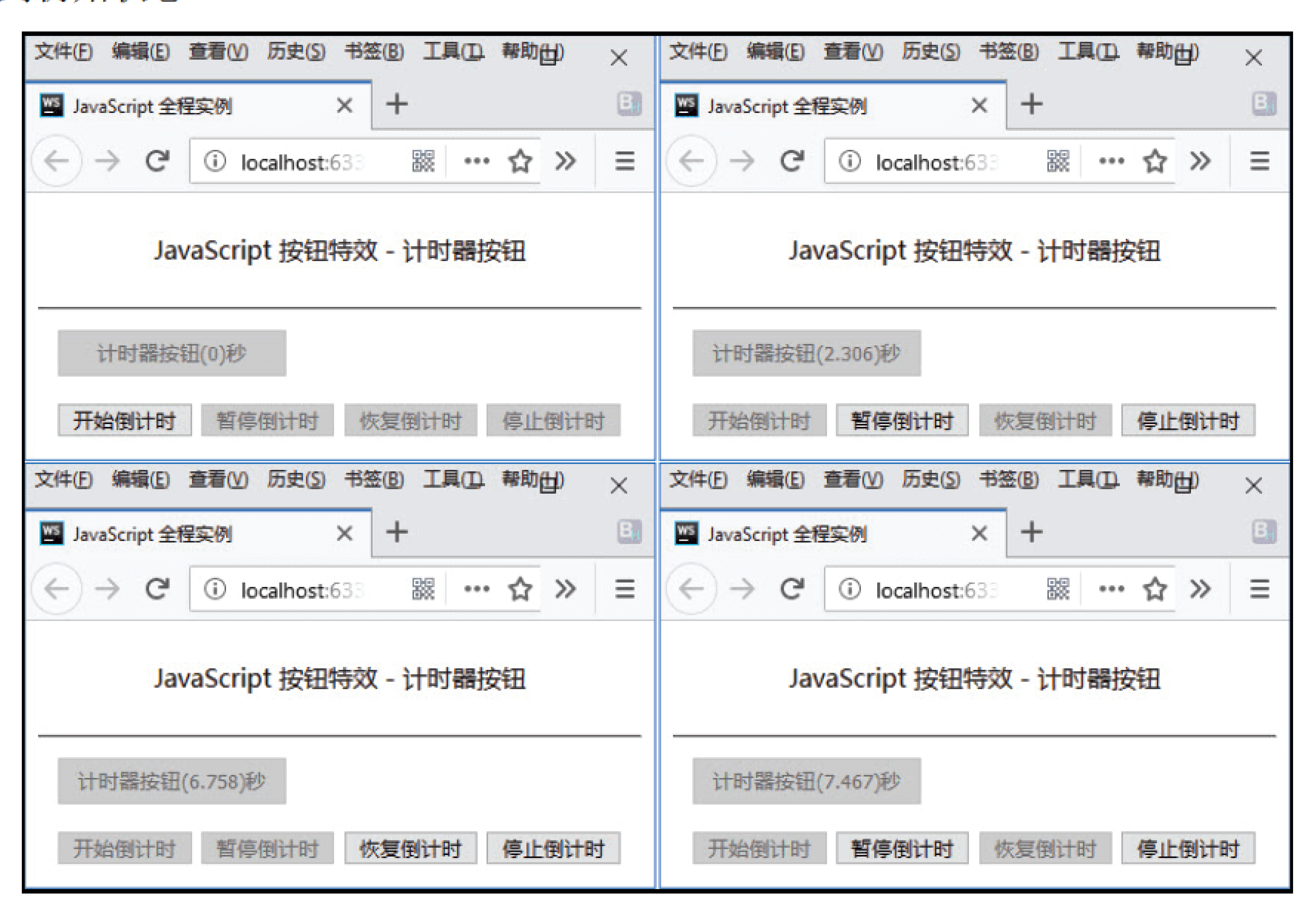


图 4.8 JavaScript 注册按钮倒计时效果

4.10 阅读完协议才可以单击的注册按钮

前面介绍了具有倒计时效果的注册按钮，说实话笔者对这个功能不是太“感冒”，因为强迫读者等待倒计时也是很让人无语的。那么有没有改进版的方案呢？答案是肯定的，还有一种是“阅读用户协议后注册按钮才被激活可以单击”的功能按钮。

一般注册页面的用户协议是放在一个文本域<textarea>控件内的，我们可以监听文本域的 onscroll 滚动事件，并借助滚动高度属性（scrollHeight）来判断用户是否阅读完文本域中用户协议，然后来激活启动用户注册按钮。

下面就具体介绍一个在阅读完用户注册协议后用户才可以单击注册按钮的 JavaScript 代码实例。

【代码 4-9】（详见源代码目录 ch04-js-btn-textarea-scroll.html 文件）

```
01 <!doctype html>
02 <html lang="en">
03 <head>
04     <!-- 添加文档头部内容 -->
05     <title>JavaScript 全程实例</title>
06 </head>
07 <body>
08 <!-- 添加文档主体内容 -->
09 <header>
10     <nav>阅读完协议才可以单击的注册按钮</nav>
11 </header>
12 <!-- 添加文档主体内容 -->
13 <p>
14     <textarea readonly=true id="id-textarea-reg"
                onscroll="on_scroll_reg(this.id)">
15         阅读完协议才可以单击的注册按钮…
16     </textarea>
17     <button id="id-btn-reg" disabled>用户注册</button>
18 </p>
19 </body>
20 <script type="text/javascript">
21     function on_scroll_reg(thisid) {
22         textareaReg = document.getElementById(thisid);
23         if ((textareaReg.scrollTop + textareaReg.clientHeight) >=
                textareaReg.scrollHeight) {
24             document.getElementById("id-btn-reg").disabled = false;
25         }
26     }
27 </script>
28 </html>
```

关于【代码 4-9】的说明：

- 第 14 ~ 16 行代码通过<textarea>标签定义了一个文本域，同时定义了 onscroll 滚动事件方法（on_scroll_reg(this.id)）。
- 第 17 行代码通过<button id="id-btn-reg">标签定义了一个注册按钮，初始状态是被禁用的。
- 第 21 ~ 26 行行代码是 on_scroll_reg()事件方法的实现，通过判断文本域的 scrollTop 属性和 clientHeight 属性之和是否大于 scrollHeight 属性得出用户是否阅读完注册协议，同时是否执行启用激活注册按钮的操作。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 4.9 所示。

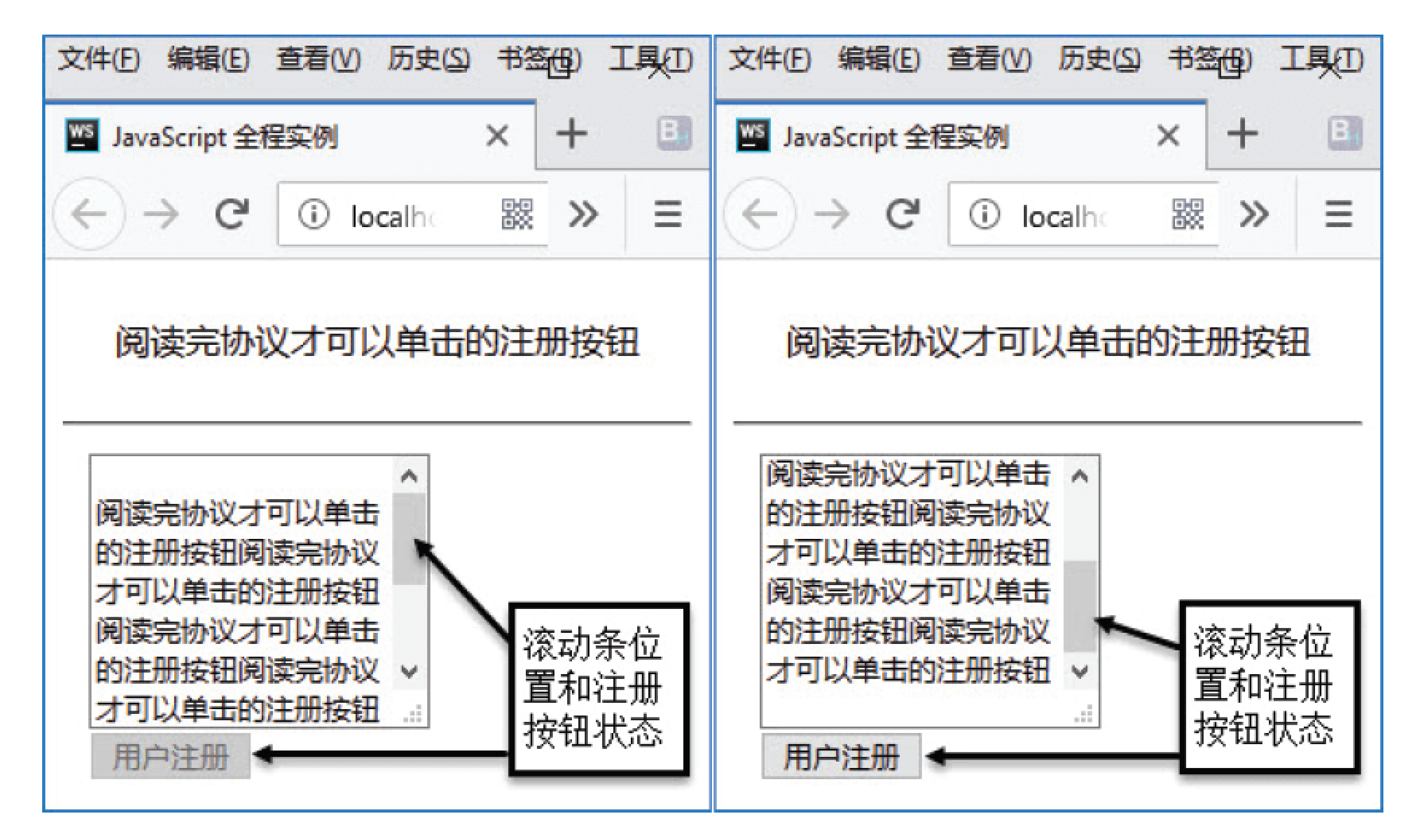


图 4.9　JavaScript 注册按钮倒计时效果

注意滚动条的初始位置以及拖动到底后注册按钮的状态（“用户注册”按钮被成功启用激活）。

第 5 章　链 接 特 效

本章介绍如何通过 JavaScript 来实现各种链接特效，通过这些特效来丰富 HTML 文档的设计手段和页面效果。

5.1　链接概述

HTML 页面中的链接用于与其他文档对象（锚点、页面和多媒体对象等）进行连接。HTML 链接（也称“超链接”）可以由一个字母、一个单词或者一句话组成，也可以是一张照片或一段视频的多媒体元素。总之，可以通过单击这些链接跳转到其他文档对象上去。

一般的，在 HTML 文档中通过使用<a>标签元素来创建链接，具体有两种定义方式：

- 通过使用 href 属性创建指向其他页面、多媒体对象的链接。
- 通过使用 name 属性创建文档内的书签（锚点）。

具体的语法格式如下：

【代码 5-1】

```
/** 通过 href 属性定义超链接 */
<a href="http://www.domain.com/" target="">打开链接</a>
/** 通过 name 属性定义书签（锚点） */
<a name="label">锚点书签（锚点）</a>
```

另外，在使用 href 属性定义 url 地址时，既可以使用绝对地址（http://），也可以使用相对地址（../..）。在使用书签（锚点）时，既可以使用 name 属性定义名称，也可以使用 id 属性定义唯一标识符。

5.2　带下划线的链接

在 HTML 页面中，超链接再被访问过后默认会带上下划线，用来标识该超链接地址已被打开过。那么，如何为未被访问过的超链接加上下划线呢？下面介绍一个通过 JavaScript 定义带下划线的超链接的代码实例。

【代码 5-2】（详见源代码目录 ch05-js-alink-underline.html 文件）

```
01 <!doctype html>
02 <html lang="en">
03 <head>
04     <!-- 添加文档头部内容 -->
05     <title>JavaScript 全程实例</title>
06 </head>
07 <style>
08     a {
09         text-decoration: none;
10     }
11 </style>
12 <body>
13 <!-- 添加文档主体内容 -->
14 <header>
15     <nav>带下划线的链接</nav>
16 </header>
17 <!-- 添加文档主体内容 -->
18 <div class="css-p">
19     <a href="#">带下划线的链接（href）</a><br/><br/>
20     <a name="#">带下划线的链接（name）</a><br/><br/>
21     <!-- 定义按钮 -->
22     <input type="button" value="为全部链接添加下划线"
              onclick="addLinkUnderline();"/>
23 </div>
24 </body>
25 <script type="text/javascript">
26     function addLinkUnderline() {
27         // TODO: 获取到所有的链接<a>
28         var aLinks = document.getElementsByTagName("a");
29         for (var i = 0; i < aLinks.length; i++) {      //遍历
30             var link = aLinks[i];   //当前的链接 DOM
31             link.style.textDecoration = 'underline';    //设置下划线样式
32         }
33     }
34 </script>
35 </html>
```

关于【代码 5-2】的说明：

- 第 19～20 行代码通过<a>标签元素定义一组链接，分别使用 href 属性和 name 属性来定义。这里需要读者注意的是，采用第 08～10 行代码中将<a>标签定义的无下划线的样式，这样页面中所显示的全部<a>标签均是无下划线的。

- 第 22 行代码通过<input>标签元素定义了一个按钮，并定义了 onclick 单击事件方法（addLinkUnderline()）。
- 第 26 ~ 33 行代码是 addLinkUnderline()事件方法的具体实现。其中，第 31 行代码通过为 Style 对象的 textDecoration 属性定义属性值（'underline'），将全部链接<a>定义为下划线样式。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 5.1 所示。

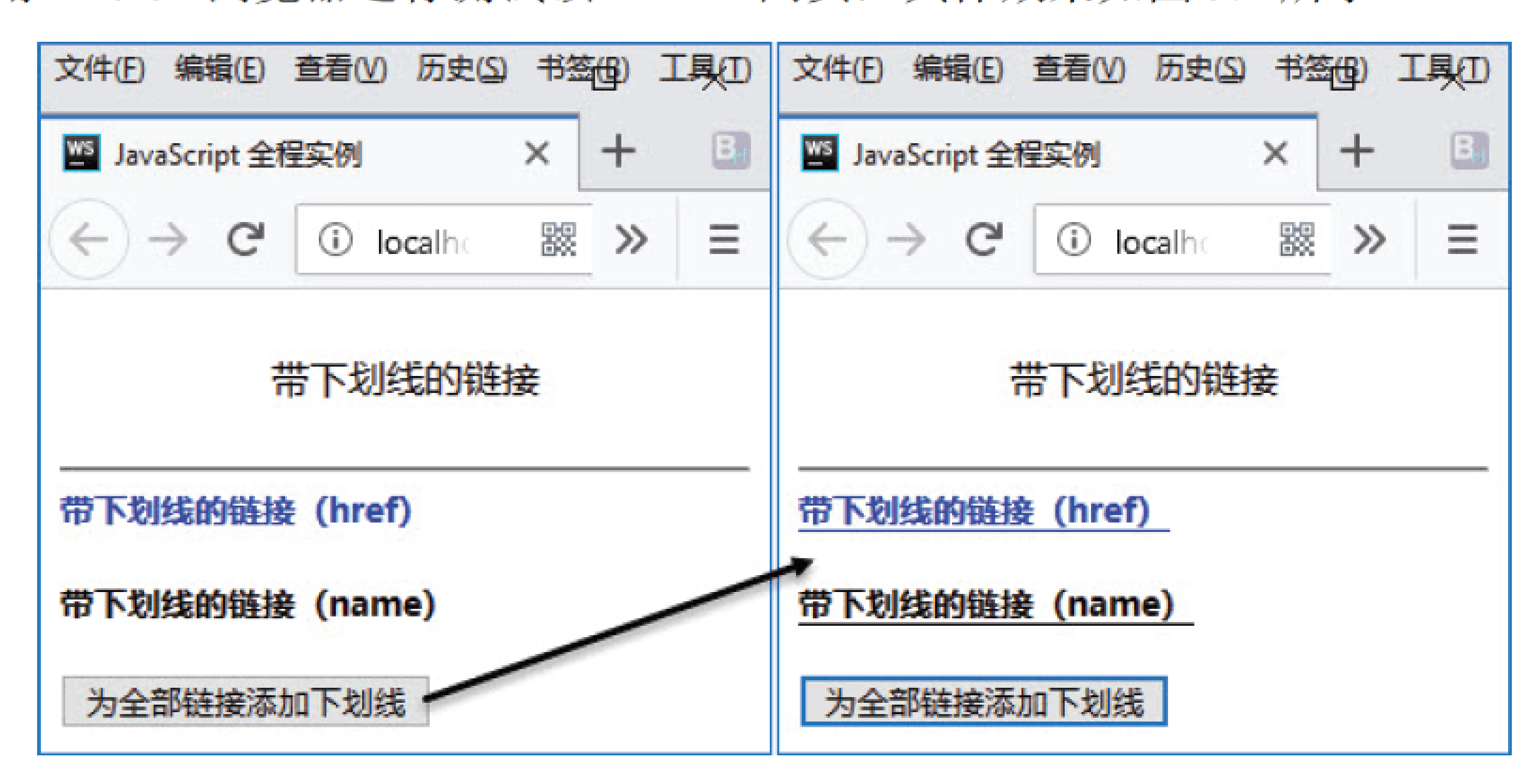


图 5.1　JavaScript 定义带下划线的链接

如图 5.1 中箭头所示，单击“为全部链接添加下划线”按钮后，页面中的全部超链接均添加上了下划线样式。

5.3　改变链接的 click 事件

对于链接<a>标签元素来讲，单击 click 操作后的跳转目标地址正常都是由 href 属性定义的 url 地址或锚点。那么，可不可以改变<a>标签元素中的 click 事件呢？下面介绍一个通过 JavaScript 改变链接 click 事件的代码实例。

【代码 5-3】（详见源代码目录 ch05-js-alink-modify-click.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>改变链接的 click 事件</nav>
</header>
<!-- 添加文档主体内容 -->
```

```
13 <div class="css-p">
14     <p>
15         <a href="#anchor" id="id-m-link">一个链接</a>
16     </p>
17     <p>
18         <a name="anchor" id="id-m-anchor">一个锚点</a>
19     </p>
20 </div>
21 </body>
22 <script type="text/javascript">
23     window.onload = function () {
24         var mLink = document.getElementById("id-m-link");
25         mLink.onclick = function () {
26             alert('超链接被点击了!');  //事件的内容逻辑
27             return false;   //返回
28         };
29     };
30 </script>
31 </html>
```

关于【代码 5-3】的说明：

- 第 15 行代码通过<a>标签元素定义了一个链接，将 href 属性值定义为一个锚点值（#anchor）、id 属性值定义为（"id-m-link"）。
- 第 18 行代码通过<a>标签元素定义了一个锚点，将 name 属性值定义为（#anchor），这样就可以保证在单击第 15 行代码定义的链接后，跳转到第 18 行代码定义的锚点位置。

下面使用 Firefox 浏览器运行测试该 HTML 网页（先注销第 23～29 行代码），具体效果如图 5.2 所示。

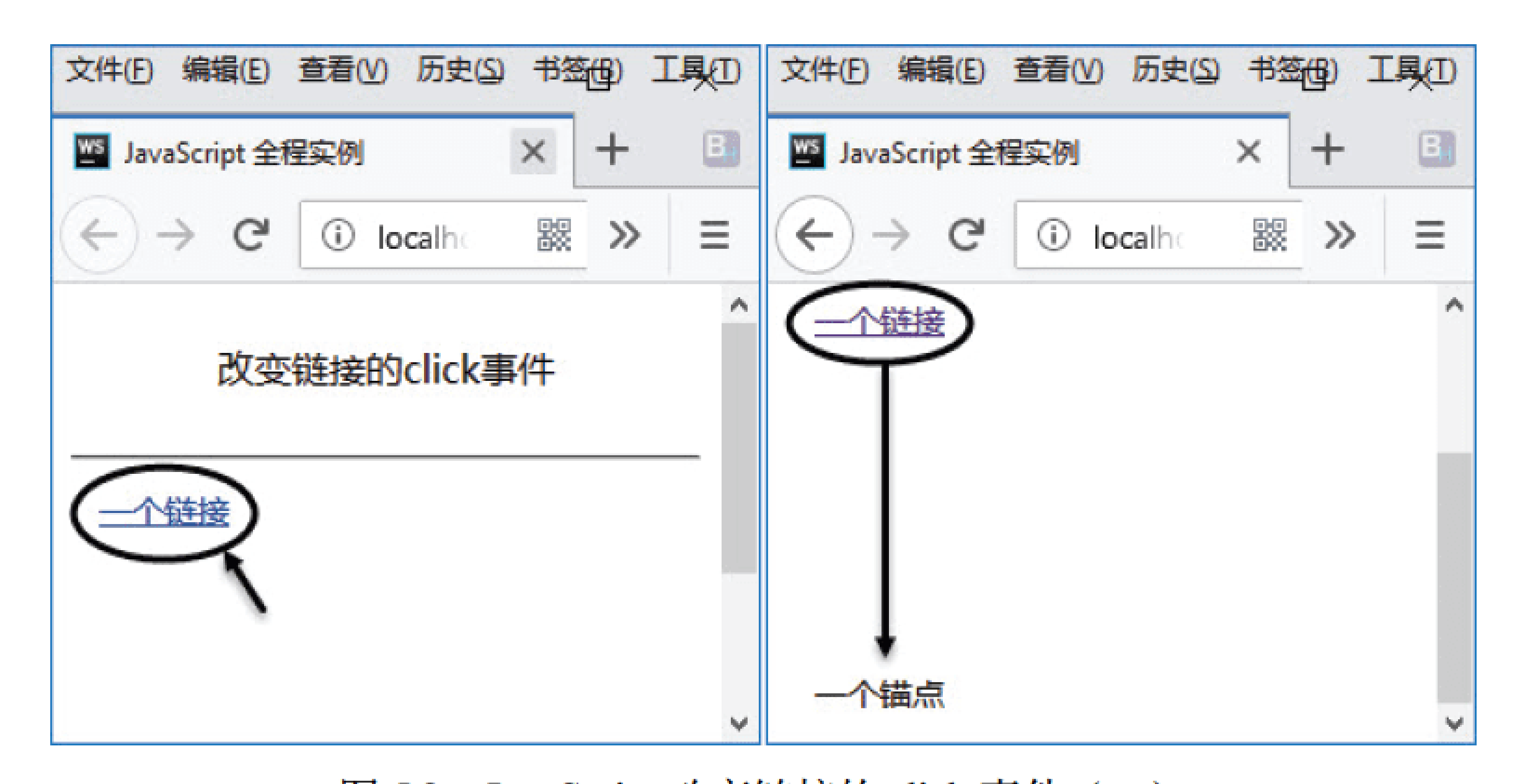


图 5.2 JavaScript 改变链接的 click 事件（一）

如图 5.2 中箭头所示，单击“一个链接”的超链接后，页面会自动跳转到“一个锚点”的锚点位置。

下面继续看【代码 5-3】的说明：

- 第 23～29 行代码定义了 Window 对象的 onload 事件处理方法。其中，第 25～28 行代码为第 15 行代码定义的超链接（id="id-m-link"）添加了 onclick 单击事件处理方法，第 26 行代码定义了一个弹出式 alert 警告框，第 27 行代码通过 return 语句返回。这样，当用户单击第 15 行代码定义的超链接时，就会执行第 25～28 行代码定义的 onclick 单击事件处理方法，而 href 属性定义的锚点地址会被覆盖掉。

下面继续使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 5.3 所示。

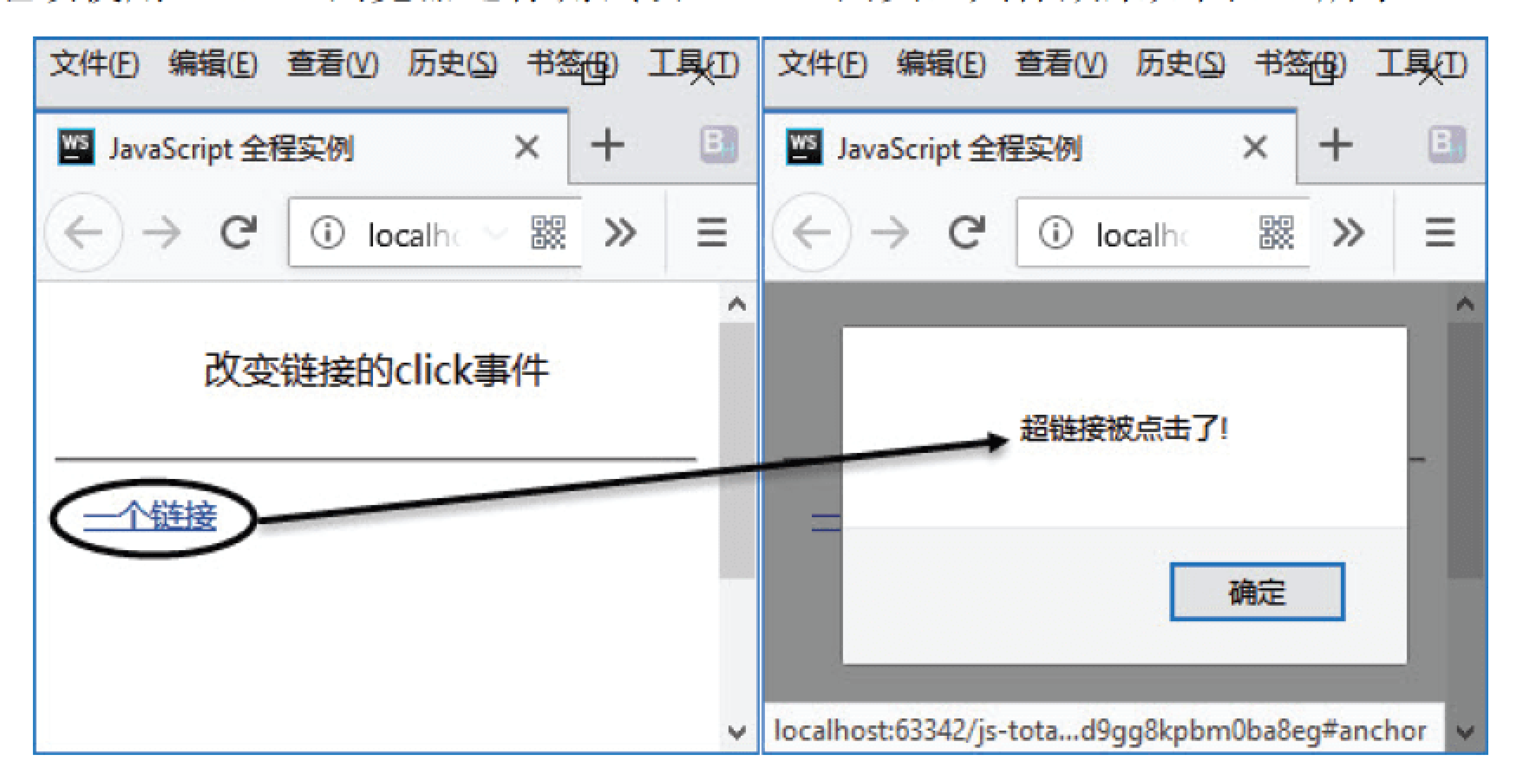


图 5.3　JavaScript 改变链接的 click 事件（二）

如图 5.3 中箭头所示，单击“一个链接”链接后，页面中会弹出一个警告框。另外，读者可以测试一下，当单击警告框中的“确定”按钮关闭该警告框后，页面是不会如图 5.2 中的效果那样自动跳转到“一个锚点”位置上的。

5.4　关闭窗口的“X”链接

在 HTML DOM 的 Window 对象中，定义有一个 close()方法可以关闭窗口。这样，设计人员就可以在页面中自定义一个“X”链接来关闭页面窗口了。需要读者注意的是，该方法对于各个浏览器的兼容性（主要是由于有的浏览器自身进行了限制）不太友好，所以设计人员在使用时要慎重。下面介绍一个通过 JavaScript 实现关闭窗口的“X”链接的代码实例。

【代码 5-4】（详见源代码目录 ch05-js-alink-x-close.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
```

```
07 <!-- 添加文档主体内容 -->
08 <header>
09     <nav>关闭窗口的“X”链接</nav>
10 </header>
11 <!-- 添加文档主体内容 -->
12 点击 <a href="#" id="id-x-link">X</a> 关闭窗口
13 </body>
14 <script type="text/javascript">
15     var xLink = document.getElementById("id-x-link");
16     xLink.onclick = function() {
17         window.close();     //关闭窗口
18         return false;       //返回
19     };
20 </script>
21 </html>
```

关于【代码 5-4】的说明：

- 第 12 行代码通过<a>标签元素定义了一个链接，风格样式接近窗口标题栏中的关闭按钮。
- 第 17 行代码通过调用 Window 对象的 close()方法关闭当前页面窗口。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 5.4 所示。

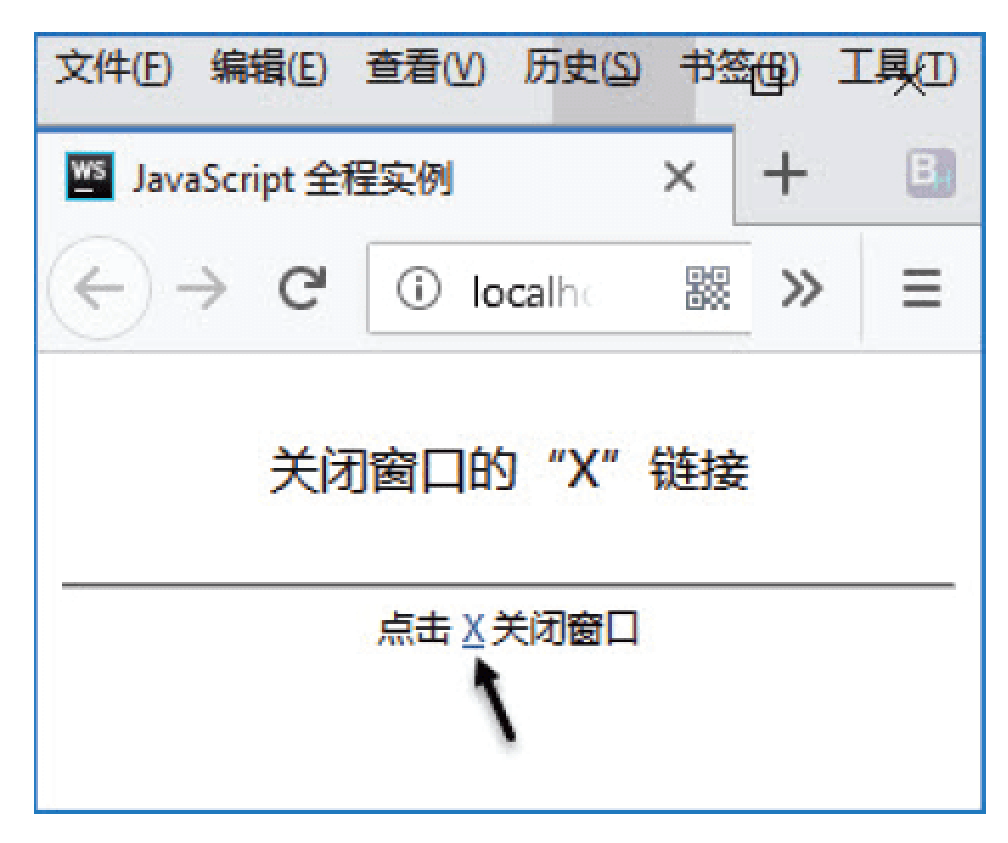


图 5.4　JavaScript 实现关闭窗口的“X”链接

如图 5.4 中箭头所示，当点击页面中“X”超链接后，会关闭当前页面窗口。

5.5　用链接模拟一个按钮

对于链接和按钮而言，从本质上虽然是两种不同的 HTML 标签元素，但最终所实现的功能从某种意义上讲是十分类似的。那么可不可以将链接模拟成为一个按钮呢？下面介绍一个通过 JavaScript 将链接模拟成为按钮的代码实例。

【代码 5-5】（详见源代码目录 ch05-js-alink-btn.html 文件）

```
01 <!doctype html>
02 <html lang="en">
03 <head>
04     <!-- 添加文档头部内容 -->
05     <title>JavaScript 全程实例</title>
06 </head>
07 <body>
08 <!-- 添加文档主体内容 -->
09 <header>
10     <nav>用链接模拟一个按钮</nav>
11 </header>
12 <!-- 添加文档主体内容 -->
13 <p>
14     <a href="#" id="id-link-btn">按钮式链接</a>
15 </p>
16 </body>
17 <script type="text/javascript">
18     window.onload = function () {
19         var linkBtn = document.getElementById("id-link-btn");
20         // TODO: 将链接定义成按钮样式
21         linkBtn.style.fontSize = '16px';          //字体
22         linkBtn.style.textAlign = 'center';       //字符对齐
23         linkBtn.style.color = 'white';            //颜色
24         linkBtn.style.padding = '4px 10px';       //内距离
25         linkBtn.style.margin = '3px';             //外边框
26         linkBtn.style.background = 'gray';       //背景色
27         linkBtn.style.textDecoration = 'none';    //文本样式
28         // TODO: 边框颜色，造成深浅效果来模仿按钮
29         linkBtn.style.borderColor = '#EEEEEE #666 #666 #EEEEEE';
30         linkBtn.style.borderStyle = 'solid';      //边框样式为实线
31         linkBtn.style.borderWidth = '1px';        //边框的粗细
32     };
33 </script>
34 </html>
```

关于【代码 5-5】的说明：

- 第 14 行代码通过<a>标签元素定义了一个链接，目标是把该链接模拟成为一个按钮的样式。
- 第 21～31 行代码通过 Style 对象将链接的样式模拟成为按钮的样式，主要是针对背景颜色（background）属性和一组边框样式（borderColor、borderStyle 和 borderWidth）属性的定义。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 5.5 所示。

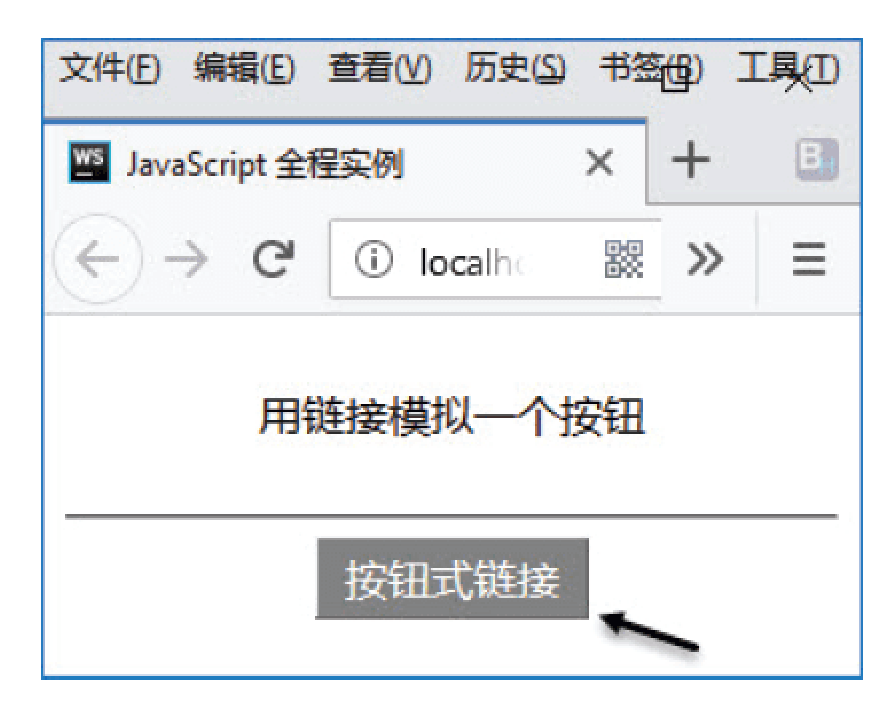


图 5.5　JavaScript 实现用链接模拟一个按钮

如图 5.5 中箭头所示，HTML 文档中原本定义的超链接完全被模拟成为一个按钮的样式了。

5.6　用链接替代表单提交按钮

5.5 节中实现了用链接模拟一个按钮样式的操作，本节将进一步用链接替代表单中的提交按钮，真正实现提交表单的功能。下面看一个通过 JavaScript 实现用链接替代表单提交按钮的代码实例。

【代码 5-6】（详见源代码目录 ch05-js-alink-submit.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>用链接替代表单提交按钮</nav>
</header>
<!-- 添加文档主体内容 -->
<form name="formLink" method="get" action="ch05-js-alink-submit.php">
    <table>
        <tr>
            <th>用户名:  </th>
            <td><input type="text" name="username"/></td>
        </tr>
        <tr>
            <th></th>
            <td>
```

```
22                  <a href="#" id="id-link-submit" onclick="on_link_submit()">
                                                                        提交表单</a>
23              </td>
24          </tr>
25      </table>
26  </form>
27  </body>
28  <script type="text/javascript">
29      /**
30       * 通过链接提交表单
31       */
32      function on_link_submit() {
33          document.formLink.submit();  //TODO: 提交表单
34      }
35  </script>
36  </html>
```

关于【代码 5-6】的说明：

- 第 13 ~ 26 行代码通过<form>标签元素定义了一个表单。其中，第 17 行代码定义了一个文本框，用于输入用户名；第 22 行代码通过<a>标签元素定义了一个链接，并定义了单击 onclick 事件方法（on_link_submit()），目的是把该链接模拟成为一个提交（Submit）按钮。
- 第 32 ~ 34 行代码是对 onclick 事件方法（on_link_submit()）的具体实现，第 33 行代码直接通过调用 Form 对象的 submit()方法实现表单的提交，这样就将链接<a>标签替代成为提交（Submit）按钮。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 5.6 所示。

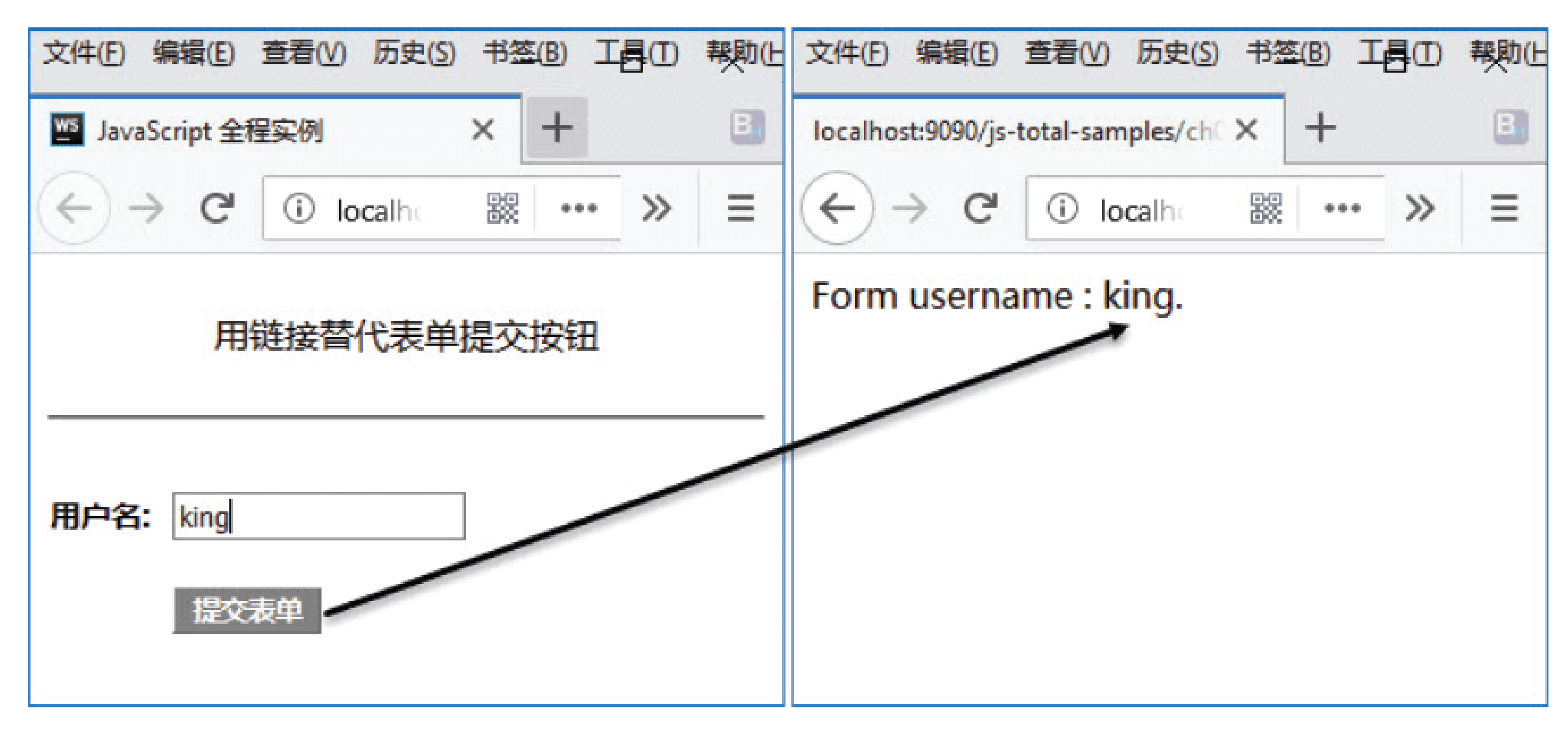


图 5.6　JavaScript 实现用链接替代表单提交按钮

如图 5.6 中箭头所示，第 22 行代码定义的链接<a>标签被模拟成为一个提交按钮样式。同时，从单击链接“提交表单”后的提交结果来看，链接<a>成功替代了表单中提交（Submit）按钮的功能。

5.7 动态修改一个链接的地址

超链接<a>标签元素中定义有一个属性 href，用于指定该链接的目标 URL 地址或锚点。可以通过动态修改该属性值达到动态修改链接目标地址的功能。下面看一个通过 JavaScript 实现动态修改一个链接地址的代码实例。

【代码 5-7】（详见源代码目录 ch05-js-alink-modify-href.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>动态修改一个链接的地址</nav>
</header>
<!-- 添加文档主体内容 -->
<p>
    <a href="#first-anchor" id="id-m-link">一个链接</a><br/>
    <input type="button" value="修改链接地址" id="id-btn"
           onclick="on_modify_url();"/>
</p>
<p>
    <a name="first-anchor">第一个锚点</a>
</p>
<p>
    <a name="second-anchor">第二个锚点</a>
</p>
</body>
<script type="text/javascript">
    /**
     * 动态修改链接的地址
     */
    function on_modify_url() {
        var mLink = document.getElementById("id-m-link");
        mLink.href = "#second-anchor";    // TODO: 修改链接的地址
    }
</script>
</html>
```

关于【代码 5-7】的说明：

- 第 18 行和第 21 行代码通过<a>标签元素定义了两个锚点（"first-anchor"和"second-anchor"）。
- 第 14 行代码通过<a id="id-m-link">标签元素定义了一个链接，初始的 href 属性值为第一个锚点（"#first-anchor"）。
- 第 15 行代码通过<input>标签定义了一个按钮，并定义了单击 onclick 事件处理方法（on_modify_url()），目标是通过动态修改<a id="id-m-link">标签元素的 href 属性值来改变链接的目标地址。
- 第 28～31 行代码是对 onclick 事件方法（on_modify_url()）的具体实现。其中，第 30 行代码直接通过修改 href 属性值，将初始的第一锚点（"#first-anchor"）改变成第二个锚点（"#second-anchor"）。

5.8 让所有链接都在新窗口打开

超链接<a>标签元素中定义有一个 target 属性，用于规定以何种方式打开链接文档。该属性的默认值为“_self”，表示在当前窗口打开文档；如果将属性值定义为“_blank”，则表示在新窗口中打开文档。

下面看一个通过 JavaScript 实现让所有链接都在新窗口打开的代码实例。

【代码 5-8】（详见源代码目录 ch05-js-alink-target-blank.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>让所有链接都在新窗口打开</nav>
</header>
<!-- 添加文档主体内容 -->
<div class="css-p">
    <!-- 定义链接 -->
    <a href="#">链接一</a><br/>
    <a href="#">链接二</a><br/>
    <a href="#">链接三</a><br/>
    <!-- 定义按钮 -->
```

```
19      <input type="button" value="修改打开方式" id="myBtn"
                onclick="modifyLinkTarget();"/>
20  </div>
21  </body>
22  <script type="text/javascript">
23      /**
24       * 设定所有链接都在新窗口打开
25       */
26      function modifyLinkTarget() {
27          //获取到所有链接的 DOM
28          var linkDOMS = document.getElementsByTagName("a");
29          for (var i = 0; i < linkDOMS.length; i++) {
30              var link = linkDOMS[i];     //当前的链接 DOM
31              link.target = "_blank";     //在新窗口中打开
32          }
33      }
34  </script>
35  </html>
```

关于【代码 5-8】的说明：

- 第 15 ~ 17 行代码通过<a>标签元素定义了一组超链接，由于未指定 target 属性值，因此 target 属性值默认为 “_self”。
- 第 29 ~ 32 行代码通过 for 循环语句，将页面中的全部链接 target 属性值修改为 “_blank”，这样再点击页面中的任何一个超链接都将会在新窗口中打开了。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 5.7 所示。

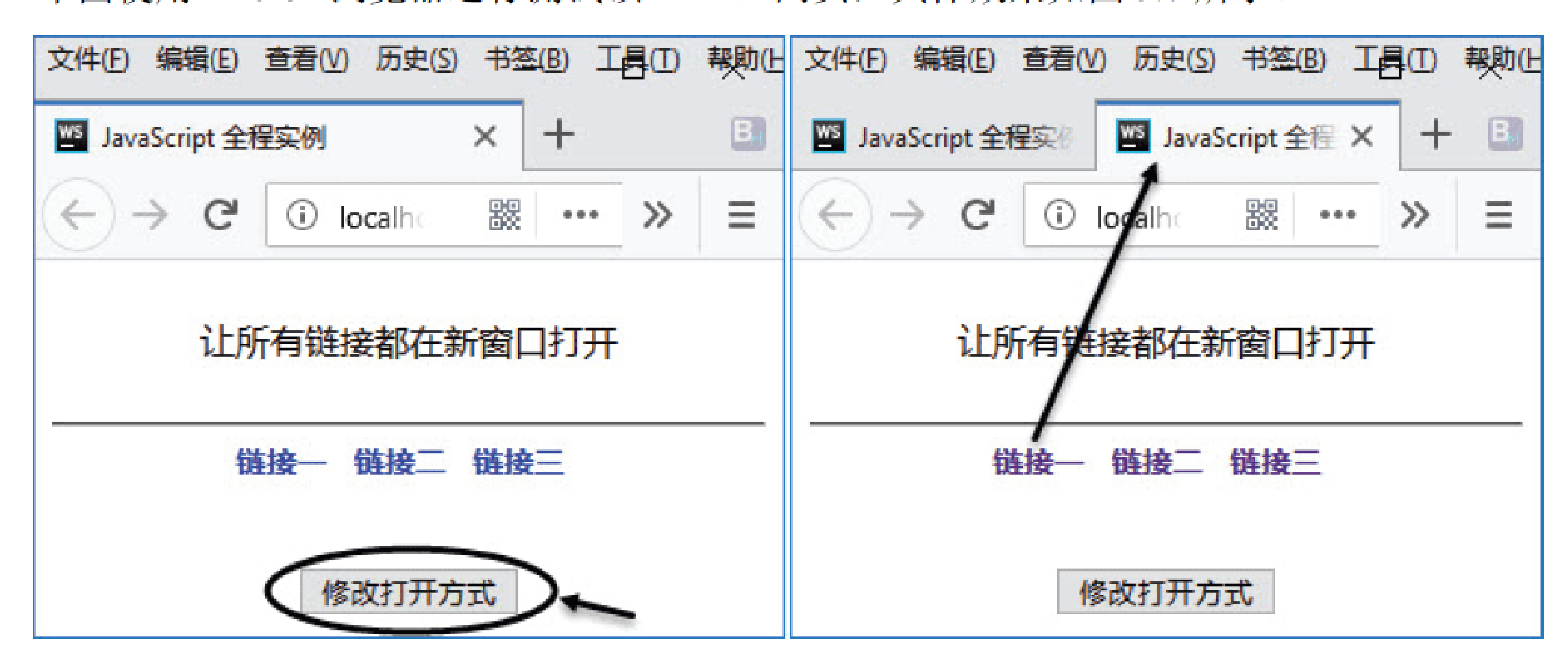


图 5.7 JavaScript 实现让所有链接都在新窗口打开

如图 5.7 中箭头所示，在通过单击“修改打开方式”按钮修改链接<a>标签元素的 target 属性值后，再单击页面中的超链接将会在新窗口中打开页面。

5.9 让页面所有的超链接都失效

超链接<a>标签元素中定义有一个 target 属性，用于规定以何种方式打开链接文档。该属性的默认值为“_self”，表示在当前窗口打开文档；如果将属性值定义为“_blank”，则表示在新窗口中打开文档。

下面看一个通过 JavaScript 实现让所有链接都失效的代码实例。

【代码 5-9】（详见源代码目录 ch05-js-alink-href-invalid.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>让页面所有的超链接都失效</nav>
</header>
<!-- 添加文档主体内容 -->
<div class="css-p">
    <!-- 定义链接 -->
    <a href="#">链接一</a>  
    <a href="#">链接二</a>  
    <a href="#">链接三</a>  <br/><br/><br/>
    <!-- 定义按钮 -->
    <input type="button" value="让超链接失效" onclick="invaildLinks();"/>
</div>
</body>
<script type="text/javascript">
    /**
     * 让页面所有的超级链接都失效
     */
    function invaildLinks() {
        //获取到所有的链接的 DOM
        var linkDOMS = document.getElementsByTagName("a");
        for (var i = 0; i < linkDOMS.length; i++) {
            var link = linkDOMS[i];     //当前的链接 DOM
            link.onclick = null;        //把单击事件置空
```

```
32             link.href = 'javascript:void(0)';   //地址改成什么都不做
33         }
34     }
35 </script>
36 </html>
```

关于【代码 5-9】的说明：

- 【代码 5-9】是在【代码 5-8】的基础上修改而成的，关键点是第 31 行和第 32 行代码。
- 第 31 行代码通过将超链接对象（link）的 onclick 事件置空，实现了屏蔽链接<a>标签元素的单击操作。
- 第 32 代码通过将超链接对象（link）的 href 属性值修改为（javascript:void(0)），表示将执行一段 js 脚本代码；而具体的脚本代码（void(0)）又是一个空方法，也就相当于让页面中的超链接都失效。

5.10　为链接地址新加一个参数

URL（统一资源定位符）是互联网上标准资源的地址，一般由页面名称和若干参数组合而成。例如“http://www.people.com/index.html?page=1&...”就是一个标准的 URL 地址写法。

超链接<a>标签元素中定义的 href 属性值就是一个具体的 URL 地址。那么，可不可以动态地为超链接<a>标签元素中定义的 URL 地址添加参数呢？下面看一个通过 JavaScript 实现为 URL 链接地址新加一个参数的代码实例。

【代码 5-10】（详见源代码目录 ch05-js-alink-href-param.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>为链接地址新加一个参数</nav>
</header>
<!-- 添加文档主体内容 -->
<div class="css-p">
    <a href="index.php">链接 1</a>  
    <a href="index.php?page=1">链接 2</a>  <br><br>
```

```
16     <input type="button" value="为链接地址新加一个参数"
               onclick="addUrlParam();"/>
17  </div>
18  </body>
19  <script type="text/javascript">
20     /**
21      * 为链接地址新加一个参数
22      */
23     function addUrlParam() {
24         //得到链接的 DOM
25         var links = document.getElementsByTagName('a');
26         //遍历所有的链接
27         for (var i = 0; i < links.length; i++) {
28             var aHref = links[i]['href'];    //得到当前的链接地址
29             //如果链接地址包含了问号，说明已经带有参数了
30             if (aHref.indexOf('?') > 0) {
31                 //使用&符号串联更多的参数
32                 links[i]['href'] = href + '&tab=2';
33             } else {
34                 //使用？来新加第一个参数
35                 links[i]['href'] = href + '?page=1';
36             }
37         }
38     }
39  </script>
40  </html>
```

关于【代码 5-10】的说明：

- 第 28 行代码通过 href 属性获取了当前链接<a>标签元素定义的 URL 地址（aHref）。
- 第 30 行代码通过 if 条件语句判断 aHref 地址是否包含字符“?”。如果包含就表示该地址已经包含参数了，执行第 32 行代码；如果不包含就表示该地址没有带参数，执行第 35 行代码。
- 第 32 行代码通过字符 '&' 串联新加的参数。
- 第 35 行代码通过字符 '?' 新加第一个参数。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 5.8 和图 5.9 所示。

如图 5.8 中箭头所示，单击“为链接地址新加一个参数”按钮后，浏览器状态栏中显示的 URL 地址新加了第 1 个参数（page=1），注意问号（?）的使用方法。

如图 5.9 中箭头所示，单击“为链接地址新加一个参数”按钮后，浏览器状态栏中显示的 URL 地址新加了第 2 个参数（tab=2），注意&符号的使用方法。

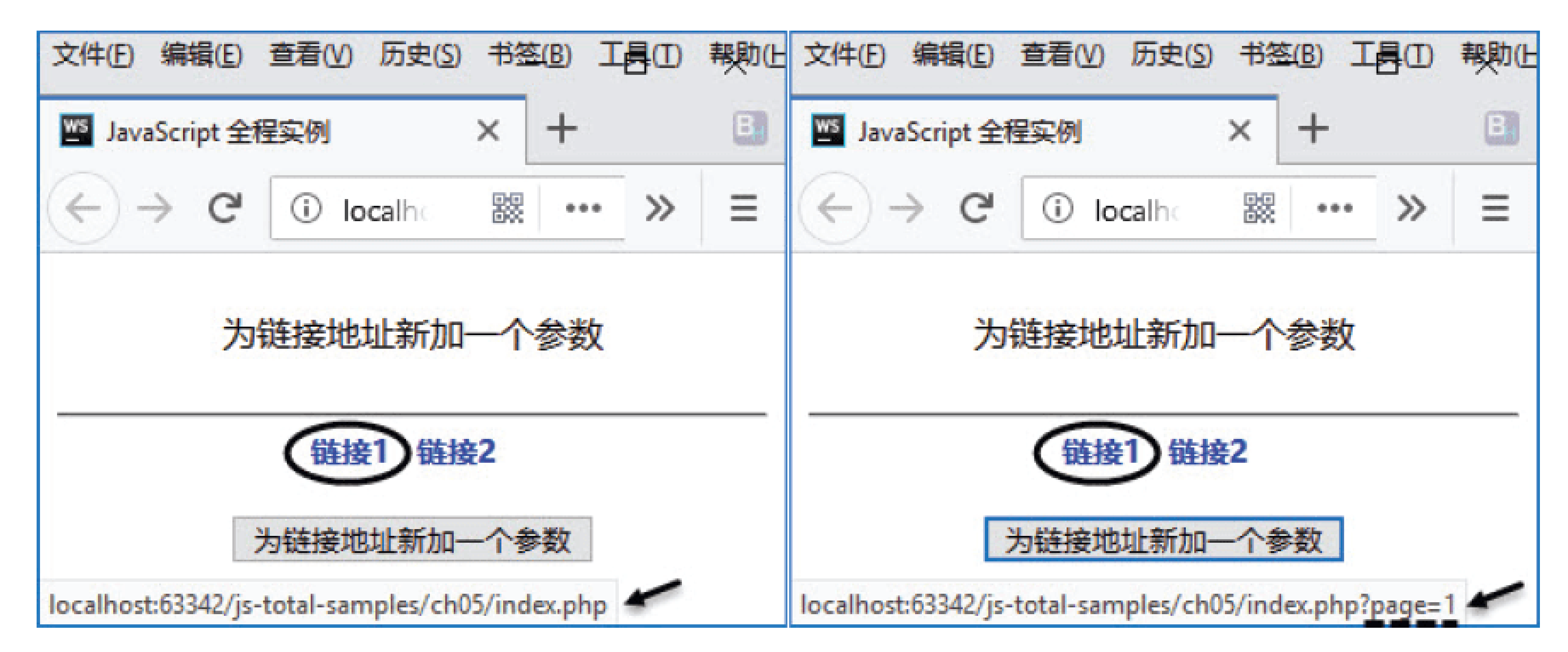


图 5.8　JavaScript 实现为链接地址新加一个参数（第一个参数的情况）

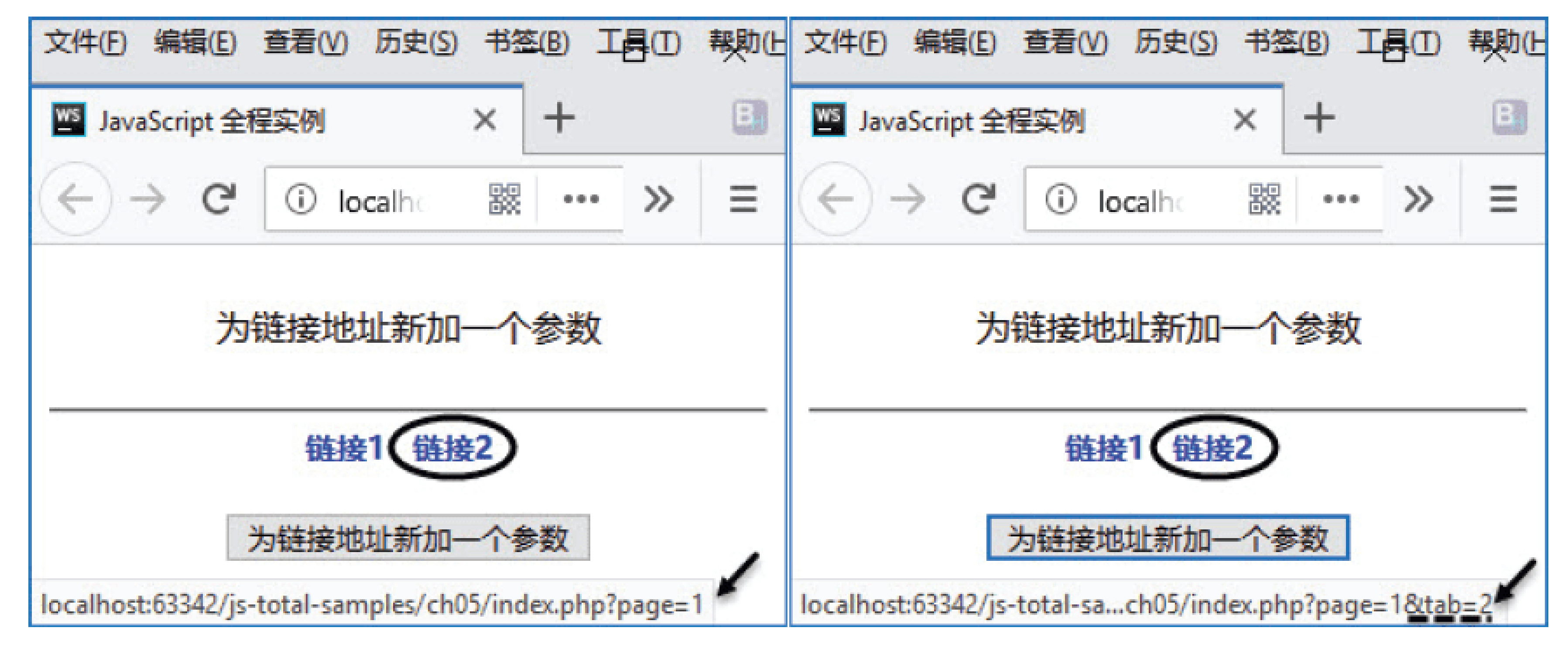


图 5.9　JavaScript 实现为链接地址新加一个参数（已带有参数的情况）

5.11　返回页面顶部的链接

相信读者在浏览很多大型门户网站的主页时都会发现页面栏目板块非常多，而且页面的长度很长。一般情况下，这类页面会在每个栏目板块内都定义一个链接（例如：返回顶部），用于执行返回到页面顶部的操作，是一个非常实用的功能。

如果要实现返回页面顶部链接的功能，就要使用到 scrollTop 这个属性。根据 HTML DOM 规范的定义，scrollTop 属性可以获取或设置一个元素的内容垂直滚动的像素数。如果我们尝试将<html>对象的 scrollTop 属性设置为 0，是不是就可以实现返回页面顶部的操作了呢？下面看一个通过 JavaScript 实现定义回到页面顶部链接的代码实例。

【代码 5-11】（详见源代码目录 ch05-js-alink-scrollTop.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
```

```
06 </head>
07 <body>
08 <!-- 添加文档主体内容 -->
09 <header>
10     <nav>JavaScript 链接特效 - 返回页面顶部的链接</nav>
11 </header>
12 <!-- 添加文档主体内容 -->
13 <a href="javascript:toPageTop();">回到顶部</a>
14 <a href="javascript:toPageTop();">回到顶部</a>
15 <a href="javascript:toPageTop();">回到顶部</a>
16 </body>
17 <script type="text/javascript">
18     /**
19      * 回到页面顶部链接的方法
20      */
21     function toPageTop() {
22         // TODO: 通过修改 document 的 scrollTop 属性值来返回到页面顶部
23         if(document.compatMode === "CSS1Compat") {
24             document.documentElement.scrollTop = 0;
25         } else if(document.compatMode === "BackCompat") {
26             document.body.scrollTop = 0;
27         } else {
28             document.documentElement.scrollTop = 0;
29         }
30     }
31 </script>
32 </html>
```

关于【代码 5-11】的说明：

- 第 13～15 行代码通过<a>标签元素定义了一组超链接（"回到顶部"），其中 href 属性值均定义为一行 js 代码（javascript:toPageTop()），当用户点击该超链接时就会调用 toPageTop()方法。
- 第 21～30 行代码是对 toPageTop()方法的实现，通过设定 document 对象的 scrollTop 属性值为 0，实现返回页面顶部的操作。

需要注意的是，第 23～29 行代码中通过 if 条件选择语句判断了 compatMode 属性的取值，然后根据结果执行了不同的 js 脚本代码。原因是对于 HTML 文档而言，是否定义 DTD 类型对于 scrollTop 属性的使用方法是不同的，而 compatMode 属性就是用于判断 HTML 文档是否定义了 DTD 类型的。

5.12 需要确认的超链接

如果想在打开一个超链接之前需要用户确认一下是否打开该链接，那么这个功能如何实现呢？下面看一个通过 JavaScript 实现需要确认的超链接的代码实例。

【代码 5-12】（详见源代码目录 ch05-js-alink-confirm.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 链接特效 - 需要确认的超链接</nav>
</header>
<!-- 添加文档主体内容 -->
<a href="http://www.gov.cn/" onclick="return(confirmOpen(this.href))">
                                                        打开超链接</a>
</body>
<script type="text/javascript">
    /**
     * 需要确认的链接
     * @param link
     * @returns {boolean}
     */
    function confirmOpen(link) {
        if (confirm('请您确认打开' + link + '网址吗?')) {
            return true;    // TODO: 返回 true
        } else {
            return false;   // TODO: 返回 false
        }
    }
</script>
</html>
```

关于【代码 5-12】的说明：

- 第 13 行代码通过<a>标签元素定义了一个超链接，其中定义了 href 属性值（"http://www.gov.cn/"）；同时，还定义了 onclick 事件处理方法（return(confirmOpen(this.href))），这行 js 代码的含义是直接通过 return 语句返回 confirmOpen()方法的返回值。
  这里需要注意的是，对于超链接<a>标签元素的单击操作而言，onclick 事件方法执行先于 href 属性定义的 URL 地址；如果 onclick 事件方法返回 false，就不会跳转到 href 属性值指定的 URL 地址上去了。
- 第 21 ~ 27 行代码是对 confirmOpen()方法的实现。其中，第 22 行代码通过调动 confirm()方法让用户选择是否跳转到指定的网址上，只有用户确认后（单击 confirm 弹出窗口中的“确定”按钮）才会继续进行跳转。

下面使用 Firefox 浏览器运行测试该 HTML 网页，页面的初始效果如图 5.10 所示。

如图 5.10 中箭头所示，单击页面中的“打开超链接”链接，页面效果如图 5.11 所示。

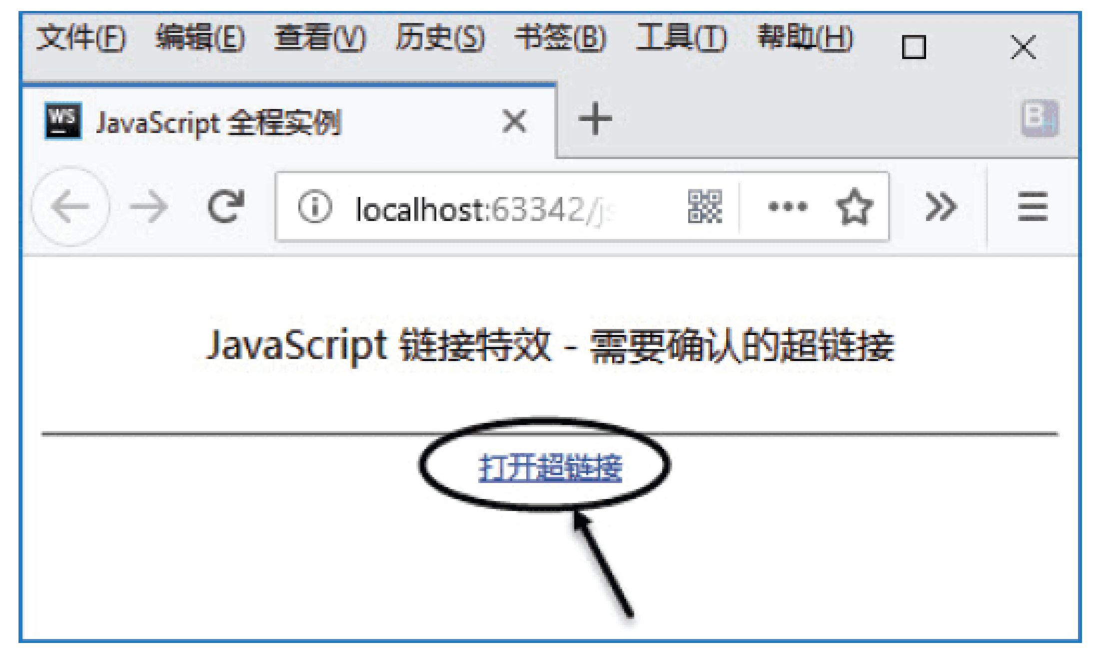


图 5.10　JavaScript 实现需要确认的超链接（初始页面）

图 5.11　JavaScript 实现需要确认的超链接（确认弹出框）

如图 5.11 中箭头所示，我们可以分别单击确认框中的“确定”按钮和“取消”按钮，页面效果如图 5.12 所示。

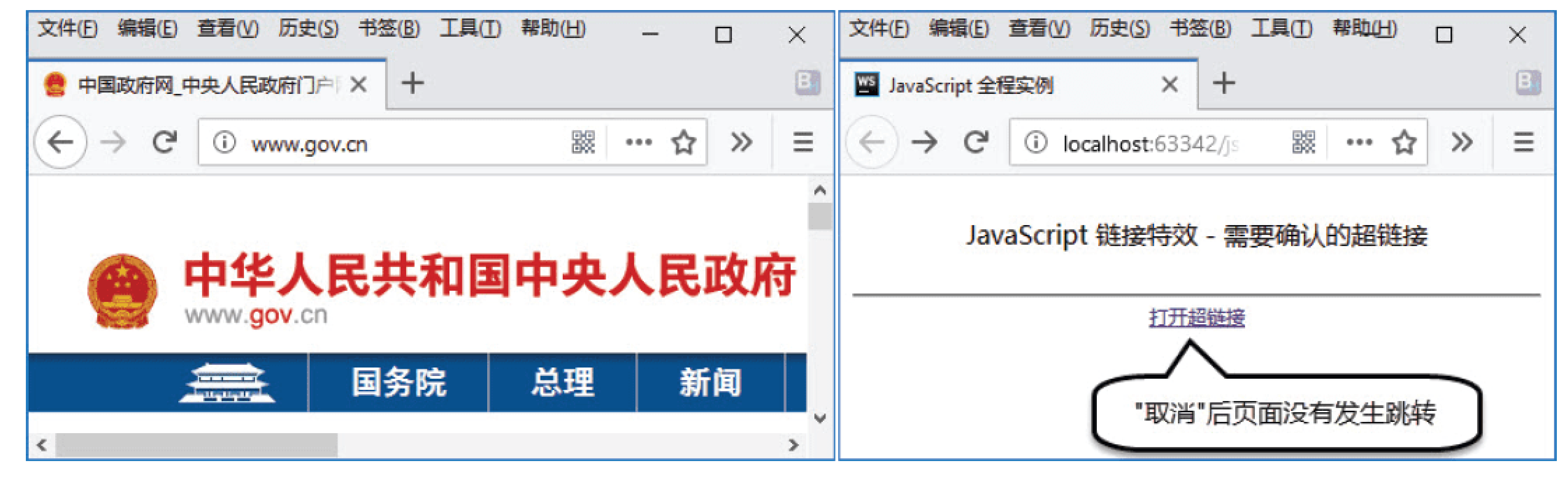


图 5.12　JavaScript 实现需要确认的超链接

第 6 章　图 片 特 效

本章介绍如何通过 JavaScript 来实现各种图片特效，通过这些特效来丰富 HTML 文档的设计手段和页面效果。

6.1　图 片 概 述

图片是 HTML 页面中必不可少的元素，可以用来丰富网页的设计手段和展示特效。在 HTML 文档中，一般通过使用<img>标签元素来定义图片，具体的语法格式如下：

【代码 6-1】

```
/** 网页图片定义 */
<img src="   "  alt="   " />
```

其中，src 属性是必需的，用于指定图像文件的 URL 地址路径；而 alt 属性是一个可选属性，用于规定在图像无法显示时的替代文本。

通过 JavaScript 脚本语言可以为网页图片实现多种特效，比如图片比例缩放、放大镜效果、自适应大小、图片边框、动态加载、延迟加载等。后面将为读者介绍多种 HTML 网页图片特效的代码实例。

6.2　图片比例缩放

在 HTML DOM 规范中，Style 对象定义有一个 transform（WebkitTransform）属性，该属性允许向元素应用旋转、缩放、移动或倾斜等 2D 变换操作。具体变换操作需要配合使用 scale(r)方法进行，该方法通过定义缩放因子（r）为当前画布添加一个缩放变换。下面介绍一个通过 JavaScript 实现图片比例缩放特效的代码实例。

【代码 6-2】（详见源代码目录 ch06-js-img-scale.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <script type="text/javascript" src="js/ch06-js-browser.js"></script>
```

```
06     <title>JavaScript 全程实例</title>
07  </head>
08  <body>
09  <!-- 添加文档主体内容 -->
10  <header>
11     <nav>图片比例缩放</nav>
12  </header>
13  <!-- 添加文档主体内容 -->
14  <p>
15     <img src="images/js.jpg" id="id-img-scale"/><br>
16     <input type="button" value="缩小" onclick="toScale(-1)"/>
17     <input type="button" value="放大" onclick="toScale(1)"/>
18  </p>
19  </body>
20  <script type="text/javascript">
21     var x = 1;  //当前的 x
22     var y = 1;  //当前的 y
23     function toScale(r) {   //改变尺寸的函数
24        //获取 DOM
25        var img = document.getElementById('id-img-scale');
26        if (r > 0) {    //向右
27           x += 0.1;   //x 坐标自加 0.1
28           y += 0.1;   //y 坐标自加 0.1
29        } else {       //向左
30           x -= 0.1;   //x 坐标自减 0.1
31           y -= 0.1;   //y 坐标自减 0.1
32        }
33        var browser = getBrowser();
34        if(browser = "FF") {
35           img.style.transform = 'scale(' + x + ',' + y + ')';
36        } else {
37           img.style.WebkitTransform = 'scale(' + x + ',' + y + ')';
38        }
39     }
40  </script>
41  </html>
```

关于【代码 6-2】的说明：

- 第 15 行代码通过<img>标签元素定义了一幅图片（src 属性定义了图片路径，并定义了 id 属性值），用于进行图片比例缩放的测试。

- 第 16 ~ 17 行代码通过<input>标签元素定义了一组按钮，并定义了 onclick 单击事件方法（toScale()），用于进行图片比例缩放操作。
- 第 21 ~ 22 行代码定义了两个变量（x、y），用于定义图片比例缩放的因子。
- 第 23 ~ 39 行代码是 toScale()事件方法的具体实现。其中，第 35 行通过 scale()方法修改 transform 属性值进行图片比例缩放操作，第 37 行代码通过 scale()方法修改 WebkitTransform 属性值进行图片比例缩放操作。

  transform 属性和 WebkitTransform 属性是为了满足浏览器兼容性而使用的。第 33 行代码通过调用一个 getBrowser()方法返回当前浏览器类型，该方法定义在第 05 行代码引用的外部 js 脚本文件中（src="js/ch06-js-browser.js"）。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 6.1 所示。

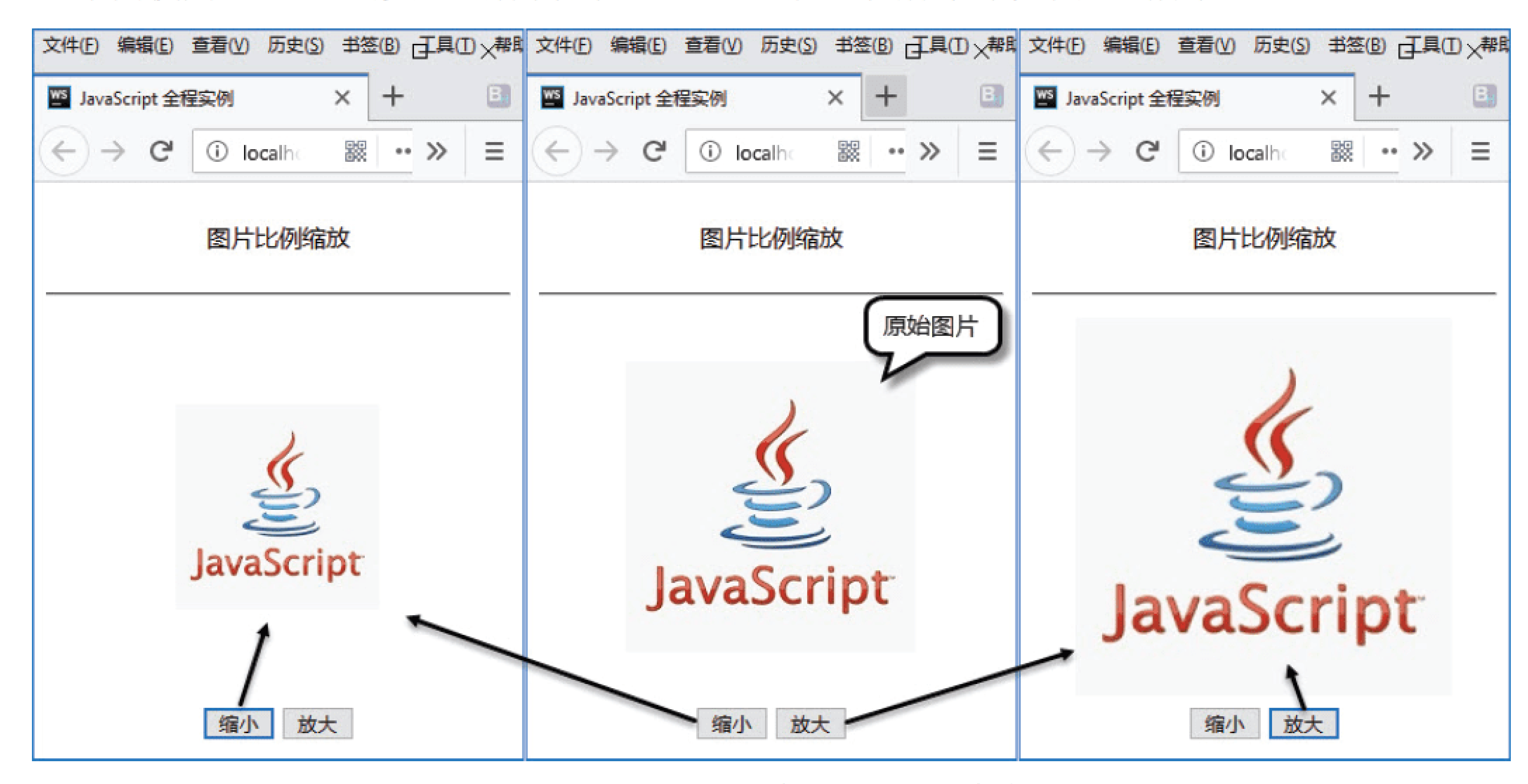


图 6.1　JavaScript 实现图片比例缩放

如图 6.1 中箭头所示，中间图片截图为原始大小，左侧图片截图为缩小后的效果，右侧图片截图为放大后的效果。

6.3　图片放大镜特效

在 HTML 网页图片设计中，还有一种非常流行的图片放大镜特效。虽然图片放大镜特效与前文中介绍的图片比例缩放特效有些类似，但是二者的实现方法还是有区别的。下面介绍一个通过 JavaScript 实现图片放大镜特效的代码实例。

【代码 6-3】（详见源代码目录 ch06-js-img-zoom.html 文件）

```
<!doctype html>
<html lang="en">
<head>
```

```
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 图片特效 - 图片放大镜特效</nav>
</header>
<!-- 添加文档主体内容 -->
<div class="small-box" id="smallBox">
    <img src="images/js-zoom.jpg"/>
    <div class="tool" id="tool"></div>
</div>
<div class="big-box" id="bigBox">
    <img src="images/js-zoom.jpg" id="bigImg"/>
</div>
</body>
<script type="text/javascript">
    var smallBox = document.getElementById("smallBox"); // TODO: smallBox
    var tool = document.getElementById("tool"); // TODO: tool
    var bigBox = document.getElementById("bigBox"); // TODO: bigBox
    var bigImg = document.getElementById("bigImg");
    /**
     * 鼠标进入 smallBox 区域内，显示 tool 区域和 bigBox 区域
     */
    smallBox.onmouseenter = function () {
        tool.className = "tool active";
        bigBox.className = "big-box active";
    };
    /**
     * 鼠标离开 smallBox 区域，不显示 tool 区域和 bigBox 区域
     */
    smallBox.onmouseleave = function () {
        tool.className = "tool";
        bigBox.className = "big-box";
    };
    /**
     * 鼠标在 smallBox 内移动
     */
    smallBox.onmousemove = function (e) {
        var _e = window.event || e;//事件对象
```

```
45         var x = _e.clientX - this.offsetLeft - tool.offsetWidth / 2;
46         var y = _e.clientY - this.offsetTop - tool.offsetHeight / 2;
47         if (x < 0) {
48             x = 0;//当左偏移出小盒子时，设为 0
49         }
50         if (y < 0) {
51             y = 0;//当上偏移出小盒子时，设为 0
52         }
53         if (x > this.offsetWidth - tool.offsetWidth) {
54             x = this.offsetWidth - tool.offsetWidth;
55         }
56         if (y > this.offsetHeight - tool.offsetHeight) {
57             y = this.offsetHeight - tool.offsetHeight;
58         }
59         tool.style.left = x + "px";
60         tool.style.top = y + "px";
61         bigImg.style.left = -x * 2 + "px";
62         bigImg.style.top = -y * 2 + "px";
63     };
64 </script>
65 </html>
```

关于【代码 6-3】的说明：

- 第 13 ~ 16 行代码通过<div>标签元素定义了一个小盒子（id="smallBox"），用于显示原始的小图片。其中，第 14 行代码通过<img>标签元素定义了该原始图片；第 15 行代码通过<div>标签元素定义了一个工具区（id="tool"），用于显示鼠标在图片上移动时的一个黄色区域。
- 第 17 ~ 19 行代码通过<div>标签元素定义了一个大盒子（id="bigBox"），用于显示放大镜图片。其中，第 18 行代码通过<img>标签元素定义了该放大的图片。
- 第 29 ~ 32 行代码定义了小盒子（id="smallBox"）的鼠标进入 onmouseenter 事件处理方法。其中，第 30 行和第 31 行代码分别激活了工具区（id="tool"）和大盒子（id="bigBox"）区域。
- 对应的，第 36 ~ 39 行代码定义了小盒子（id="smallBox"）的鼠标离开 onmouseleave 事件处理方法。其中，第 37 行和第 38 行代码分别禁用了工具区（id="tool"）和大盒子（id="bigBox"）区域。
- 第 43 ~ 63 行代码定义了小盒子（id="smallBox"）的鼠标移动 onmousemove 事件处理方法，通过窗口客户区坐标（clientX 和 clientY）属性、相对偏移量（offsetLeft 和 offsetTop）属性和控件宽高尺寸（offsetWidth 和 offsetHeight）属性，计算出工具区（id="tool"）相对小盒子（id="smallBox"）的位置，并在大盒子（id="bigBox"）中显示图片放大镜特效。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 6.2 所示。

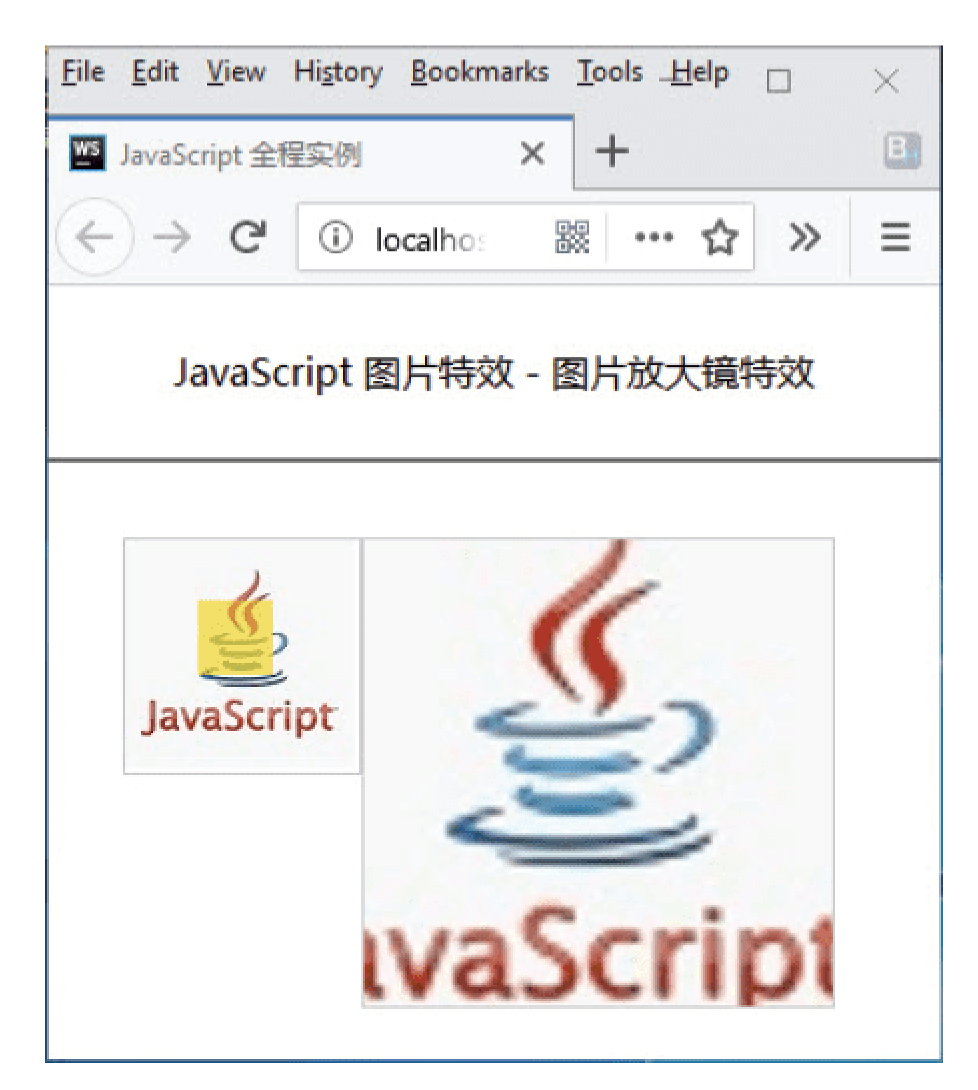


图 6.2　JavaScript 实现图片放大镜特效

6.4　图片在层里居中

在一些 HTML 网页图片设计的场景中，将图片设置为居中显示也是很常见的。获取图片尺寸属性并通过计算得出图片在父级层元素中的定位，实现图片在层里居中显示的效果。下面介绍一个通过 JavaScript 实现让图片在层里居中显示的代码实例。

【代码 6-4】（详见源代码目录 ch06-js-img-div-center.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 图片特效 - 图片在层内居中</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center">
    <img src="images/js-center.jpg" id="id-img"/>
</div>
</body>
<script type="text/javascript">
    window.onload = function () {
```

```
19          var imgCenter = document.getElementById('id-img');
20          var width = imgCenter.width;
21          var height = imgCenter.height;
22          var divCenter = imgCenter.parentNode;
23          var widthOffset = divCenter.offsetWidth;
24          var heightOffset = divCenter.offsetHeight;
25          imgCenter.style.margin = '0 auto';
26          var padding = (heightOffset - height) / 2;
27          divCenter.style.paddingTop = (32 + padding)+'px';
28      }
29  </script>
30  </html>
```

关于【代码 6-4】的说明：

- 第 13 ~ 15 行代码通过<div>标签元素定义了一个层（id="id-div-center"）。其中，第 14 行代码通过<img id="id-img">标签元素定义了一幅图片，目的是让该图片在层（id="id-div-center"）内居中显示。
- 第 18 ~ 28 行代码实现了在页面初始化过程中，让图片在层内居中显示的效果；具体是通过获取图片（id="id-img"）的宽高尺寸属性（width 和 Height）数值和层（id="id-div-center"）宽高尺寸属性（offsetWidth 和 offsetHeight）数值，计算得出图片在层内居中显示的相对位置。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 6.3 所示。

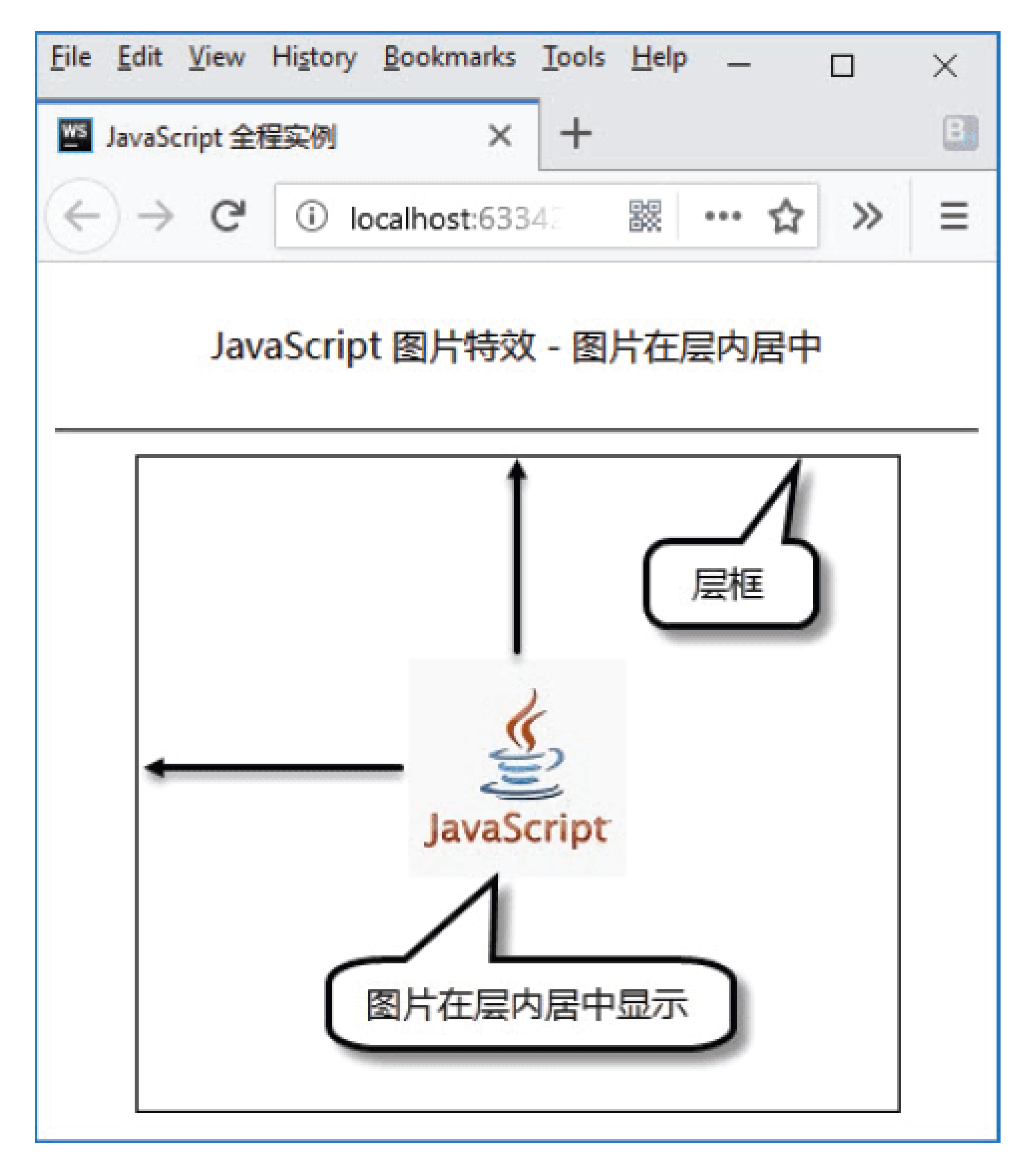


图 6.3　JavaScript 实现图片在层里居中

如图 6.3 中箭头所示，图片在外层层框（黑实线）内的显示位置是居中的。

6.5 让图片自适应框的大小

在 HTML 网页设计中，除了 6.4 节介绍的图片居中显示效果，还有一种让图片自适应框的大小也很常见。下面介绍一个通过 JavaScript 实现让图片自适应框的大小的代码实例。

【代码 6-5】（详见源代码目录 ch06-js-img-div-adjust.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 图片特效 - 图片自适应框的大小</nav>
</header>
<!-- 添加文档主体内容 -->
<img src="images/js-adjust.jpg"/>
<div id="id-div-adjsut1">
    <img src="images/js-adjust.jpg" id="id-img1"/>
</div>
<div id="id-div-adjsut2">
    <img src="images/js-adjust.jpg" id="id-img2"/>
</div>
</body>
<script type="text/javascript">
    window.onload = function () {
        var img1 = document.getElementById('id-img1');
        adjsutImgFitDiv(img1);
        var img2 = document.getElementById('id-img2');
        adjsutImgFitDiv(img2);
    };
    /**
     * Adjust img to fit div
     * @param img
     */
    function adjsutImgFitDiv(img) {
        var div = img.parentNode;
        var w = div.offsetWidth;
```

```
35          var h = div.offsetHeight;
36          img.width = w - 2;
37          img.height = h - 2;
38      }
39  </script>
40  </html>
```

关于【代码 6-5】的说明：

- 第 13 行代码通过<img>标签元素定义了一幅图片，用于显示原始图片。
- 第 14 ~ 16 行代码通过<div id="id-div-adjsut1">标签元素定义了一个放大的层框（id="id-div-adjsut1"）。其中，第 15 行代码通过<img id="id-img1">标签元素引用了上面的原始图片，目的是让图片自适应这个放大的层框。
- 相应的，第 17 ~ 19 行代码通过<div id="id-div-adjsut2">标签元素定义了一个缩小的层框（id="id-div-adjsut2"）。其中，第 18 行代码通过<img id="id-img1">标签元素同样引用了上面的原始图片，目的是让图片自适应这个缩小的层框。
- 第 32 ~ 38 行代码实现了一个自定义函数（adjsutImgFitDiv()），用于实现让图片自适应层框的算法。其中，第 34 ~ 35 行代码通过获取层框（id="id-div-center"）宽高尺寸属性（offsetWidth 和 offsetHeight）数值，重新设定图片的宽高尺寸（注意尺寸的小幅修正量）。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 6.4 所示。



图 6.4　JavaScript 实现让图片自适应框的大小

如图 6.4 中的标识所示，页面最上面是原始图片，中间是图片自适应放大层框的效果，下面是图片自适应缩小层框的效果。

6.6 为图片加上边框

在 HTML 网页设计中，可以为图片自定义边框样式，主要是边框的颜色和厚度。下面介绍一个通过 JavaScript 实现为图片加上边框的代码实例。

【代码 6-6】（详见源代码目录 ch06-js-img-border.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 图片特效 - 为图片加上边框</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-border">
    边框颜色:  
    <select id="id-sel-color" onchange="on_sel_color_changed(this.id);">
        <option value="none" selected>None</option>
        <option value="black">Black</option>
        <option value="red">Red</option>
        <option value="blue">Blue</option>
    </select>
    边框厚度:  
    <select id="id-sel-width" onchange="on_sel_width_changed(this.id);">
        <option value="0" selected>0</option>
        <option value="1">1</option>
        <option value="2">2</option>
        <option value="3">3</option>
    </select>
    <img src="images/js-border.jpg" id="id-img"/>
</div>
</body>
<script type="text/javascript">
    /**
     * select image border color
     * @param thisid
```

```
35         */
36        function on_sel_color_changed(thisid) {
37            var selColor = document.getElementById(thisid);
38            var color = selColor.value;
39            var selWidth = document.getElementById('id-sel-width');
40            var width = selWidth.value;
41            setImageBorder(width, color);
42        }
43        /**
44         * select image border width
45         * @param thisid
46         */
47        function on_sel_width_changed(thisid) {
48            var selWidth = document.getElementById(thisid);
49            var width = selWidth.value;
50            var selColor = document.getElementById('id-sel-color');
51            var color = selColor.value;
52            setImageBorder(width, color);
53        }
54        /**
55         * set image border style
56         * @param width
57         * @param color
58         */
59        function setImageBorder(width, color) {
60            var img = document.getElementById("id-img");
61            if(width == "0" || color == "none") {
62                img.style.border = "";
63            } else {
64                img.style.border = width + 'px solid ' + color;
65            }
66        }
67    </script>
68    </html>
```

关于【代码 6-6】的说明：

- 第 15～20 行和第 22～27 行代码分别通过<select>标签元素定义了两个下拉列表框，并添加了 onchange 事件处理方法，用于选择图片边框颜色和厚度。
- 第 28 行代码通过<img id="id-img">标签元素定义了一幅图片，用于测试为图片添加边框的操作。
- 第 36～42 行和第 47～53 行代码分别实现了边框颜色和边框厚度这两个下拉列表框的 onchange 事件处理方法，用于获取用户选择的颜色值和厚度值，然后通过调用自定义函数 setImageBorder()来设定图片边框的颜色和厚度样式。

- 第 59 ~ 66 行代码是自定义函数 setImageBorder()的实现过程，先判断用户选择的边框颜色或厚度是否有效（只要厚度为"0"或颜色为"none"这两个条件有一个成立，就表示图片边框样式无效），再通过设定图片 Style 对象的 border 属性来改变图片颜色和边框的样式。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 6.5 所示。

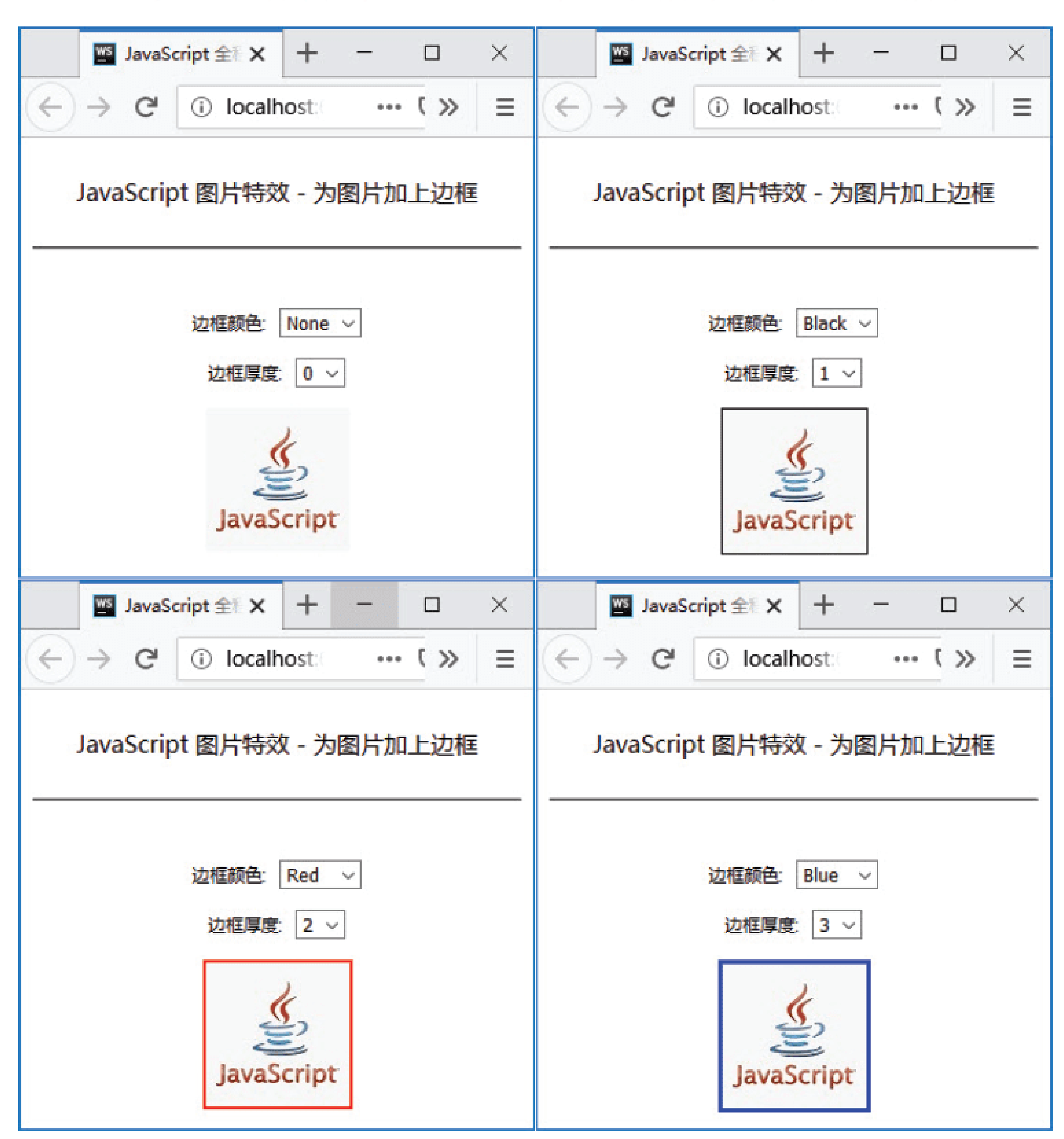


图 6.5　JavaScript 实现为图片加上边框

图 6.5 中的 4 个页面依次显示了图片边框在不同颜色值和不同厚度值下的效果。

6.7　显示局部图片

其实每一幅图片在 HTML 页面窗口中都是有定位尺寸的，通过对该定位尺寸进行变换计算，就可以选取图片的某一个局部，并在页面中进行显示。下面看一个通过 JavaScript 实现显示局部图片的代码实例。

【代码 6-7】（详见源代码目录 ch06-js-img-part.html 文件）

```
<!doctype html>
<html lang="en">
<head>
```

```
04      <!-- 添加文档头部内容 -->
05      <title>JavaScript 全程实例</title>
06  </head>
07  <style type="text/css">
08      div#id-div-img {
09          background: url('images/js-part.jpg') 0 0 no-repeat;
10      }
11  </style>
12  <body>
13  <!-- 添加文档主体内容 -->
14  <header>
15      <nav>JavaScript 图片特效 - 显示局部图片</nav>
16  </header>
17  <!-- 添加文档主体内容 -->
18  <div id="id-div-center">
19      <input type="button" value="切换显示局部图片"
                onclick="showImagePart();"/>
20  </div>
21  <div id="id-div-img"></div>
22  </body>
23  <script type="text/javascript">
24      var arr = [[0, 0], [-150, 0], [-150, -150], [0, -150]];
25      var index = 0;
26      function showImagePart() {
27          var img = document.getElementById('id-div-img');
28          index++;
29          if (index == 4)
30              index = 0;
31          var x = arr[index][0] + 'px';
32          var y = arr[index][1] + 'px';
33          img.style.backgroundPosition = x + ' ' + y;
34      }
35  </script>
36  </html>
```

关于【代码 6-7】的说明：

- 第 21 行代码通过<div id="id-div-img">标签元素定义了一个层，该元素看似是一个空层，其实不然。请读者仔细看第 08～10 行代码定义了的 CSS 样式代码，第 09 行代码通过背景 background 样式属性为层<div id="id-div-img">元素定义了背景图片。
- 第 19 行代码通过<input>标签元素定义了一个按钮，并定义了单击 onclick 事件处理方法（showImagePart()），用于测试“切换显示局部图片”的操作。
- 第 24 行代码定义了一个数组坐标（arr），标记局部图片的定位坐标。

- 第 25 行代码定义了一个变量（index），用于数组坐标（arr）下标的索引。
- 第 26 ~ 34 行代码是事件处理方法（showImagePart()）的实现过程，通过依次将数组坐标（arr）项的值赋给图片 Style 对象的 backgroundPosition 属性来实现切换显示局部图片的操作。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 6.6 所示。

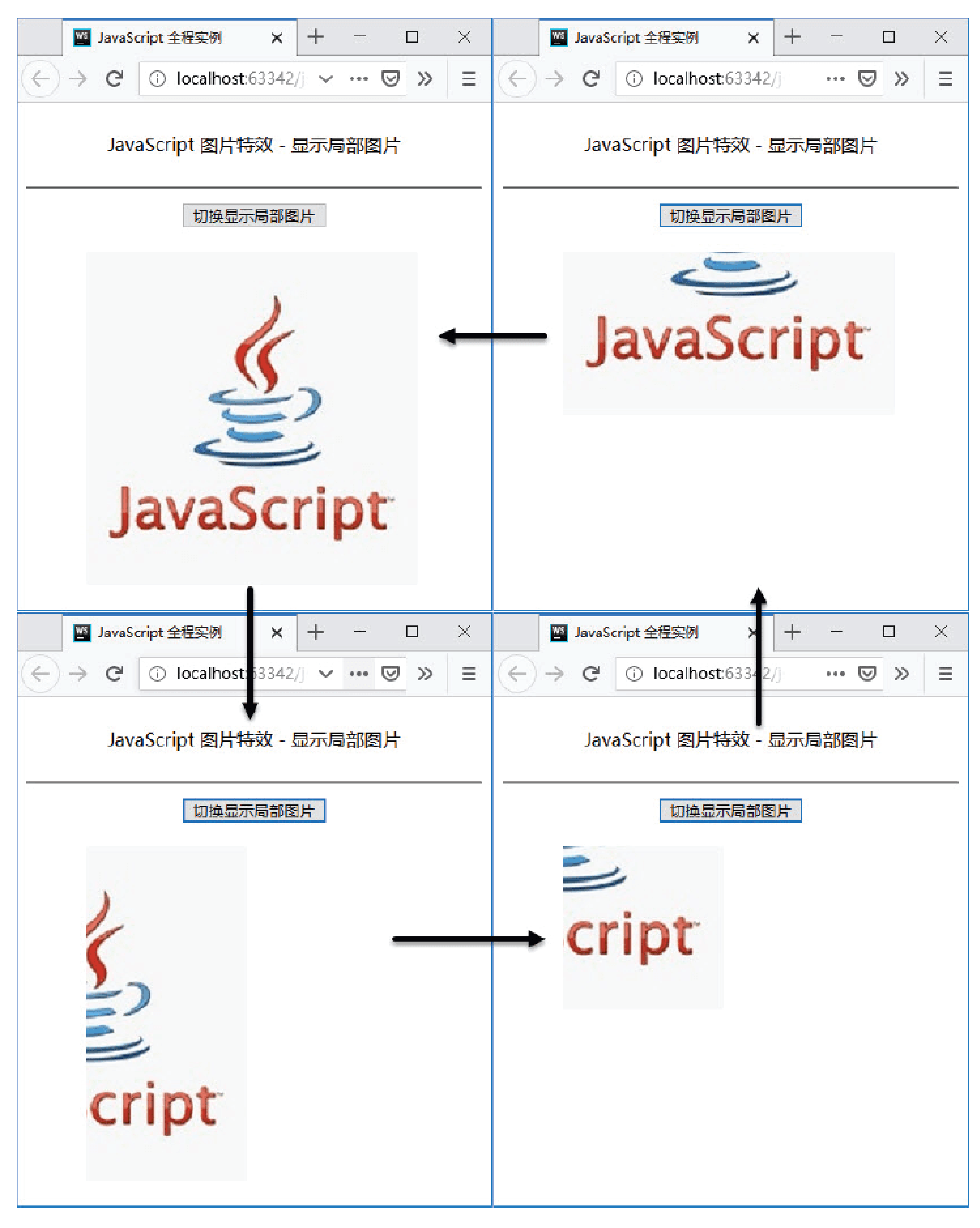


图 6.6　JavaScript 实现切换显示局部图片

如图 6.6 中箭头方向所示，页面中分别演示了“切换显示局部图片”操作的效果。

6.8　动态加载图片

在 HTML 网页图片设计中，动态加载图片是一项很实用的技术。在大型项目开发中，网页图片资源的数量往往是很庞大的，如果图片全部采用静态加载的方式是非常耗费网络资源的，而且用户体验也会很糟糕。通过动态加载图片的方式，在实际需要时将图片进行加载显示就很合理，用户体验也会得到提升。下面看一个通过 JavaScript 实现动态加载图片的代码实例。

【代码 6-8】(详见源代码目录 ch06-js-img-dyn-load.html 文件)

```
01 <!doctype html>
02 <html lang="en">
03 <head>
04     <!-- 添加文档头部内容 -->
05     <title>JavaScript 全程实例</title>
06 </head>
07 <body>
08 <!-- 添加文档主体内容 -->
09 <header>
10     <nav>JavaScript 图片特效 - 动态加载图片</nav>
11 </header>
12 <!-- 添加文档主体内容 -->
13 <div id="id-div-center">
14     <input type="button" value="动态加载图片"
           onclick="dynLoadImg(this.id);"/>
15     <img lazy_src="images/js-dyn-load.jpg" src="" alt="图片未加载"
              id="id-img"/>
16 </div>
17 </body>
18 <script type="text/javascript">
19     /**
20      * dynamic load image
21      * @param thisid
22      */
23     function dynLoadImg(thisid) {
24         var img = document.getElementById('id-img');
25         img.src = img.attributes['lazy_src'].value;
26     }
27 </script>
28 </html>
```

关于【代码 6-8】的说明：

- 第 14 行代码通过<input>标签元素定义了一个按钮，并定义了单击 onclick 事件处理方法（dynLoadImg()），用于测试"动态加载图片"的操作。
- 第 15 行代码通过<img>标签元素定义的图片是关键所在，注意其 src 属性值为空（页面初始化加载时是无图片的），而增加了一个自定义属性 lazy_src，其属性值引用了想要动态加载的图片路径。
- 第 23～26 行代码是事件处理方法（dynLoadImg()）的实现过程，通过图片对象的 attributes 属性获取自定义属性 lazy_src 的值，然后赋给 src 属性，从而实现动态加载图片的操作。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 6.7 所示。

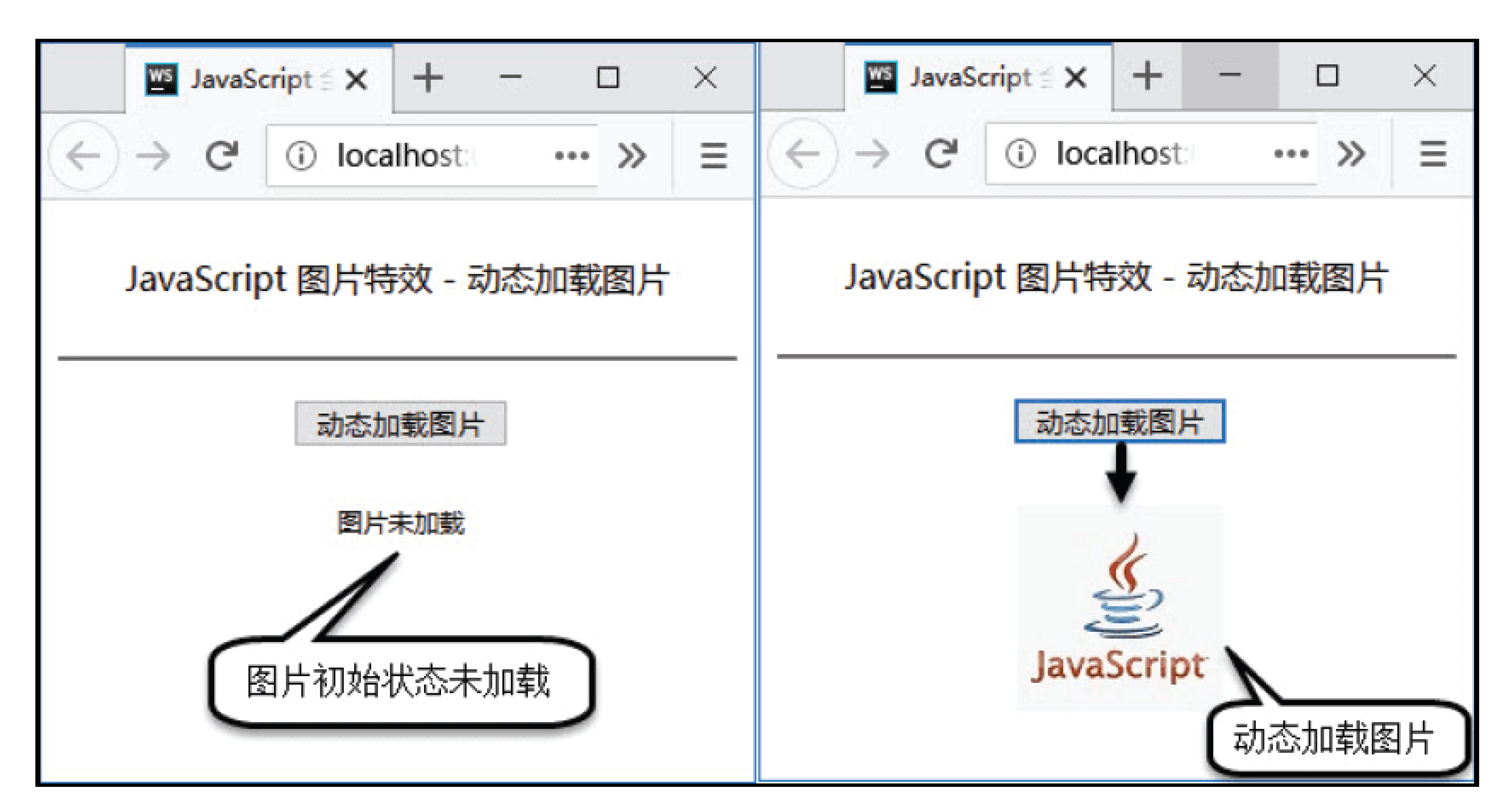


图 6.7　JavaScript 实现动态加载图片

6.9　延迟加载图片

6.8 节中介绍的动态加载图片是一项很有用的技术，不过仍需要人工来操作，不太适用于页面的加载过程。在网页图片资源的数量很庞大、要避免全部采用静态加载方式引起阻塞的情况下，可以通过自动延迟加载图片的方式来解决问题。具体来讲，就是通过计时器设定时间延迟，然后使用动态加载技术来完成图片的加载。下面看一个通过 JavaScript 实现延迟加载图片的代码实例。

【代码 6-9】（详见源代码目录 ch06-js-img-delay-load.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 图片特效 - 延迟加载图片</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center">
    <img lazy_src="images/js-delay-load.jpg" src="" alt="图片正在加载..."
            id="id-img"/>
</div>
</body>
<script type="text/javascript">
```

```
18      var timer = null;   // TODO: 计时器
19      window.onload = function (ev) {
20          timer = setTimeout(image_delay_load, 3000);
21      };
22      /**
23       * image delay load
24       */
25      function image_delay_load() {
26          var img = document.getElementById('id-img');
27          img.src = img.attributes['lazy_src'].value;
28          timer = null;
29      }
30  </script>
31  </html>
```

关于【代码 6-9】的说明：

- 第 14 行代码通过<img>标签元素定义了一幅图片，用于测试延迟加载操作，其 src 属性值为空（页面初始化加载时是无图片的），同时增加了一个自定义属性 lazy_src，其属性值引用了想要延迟加载的图片路径。
- 第 18 行代码定义了一个变量（timer），作为计时器使用。
- 第 20 行代码在页面加载过程中，通过 setTimeout()方法启动了一个计时器（timer），回调方法为 image_delay_load，延迟时间设定为 3000ms。
- 第 25～29 行代码是计时器回调方法(image_delay_load)的实现过程，通过图片对象的 attributes 属性获取自定义属性 lazy_src 的值，然后赋给 src 属性；计时器回调方法会在设定的延迟时间（3000ms）完成后被调用，从而实现了延迟加载图片的操作。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 6.8 所示。

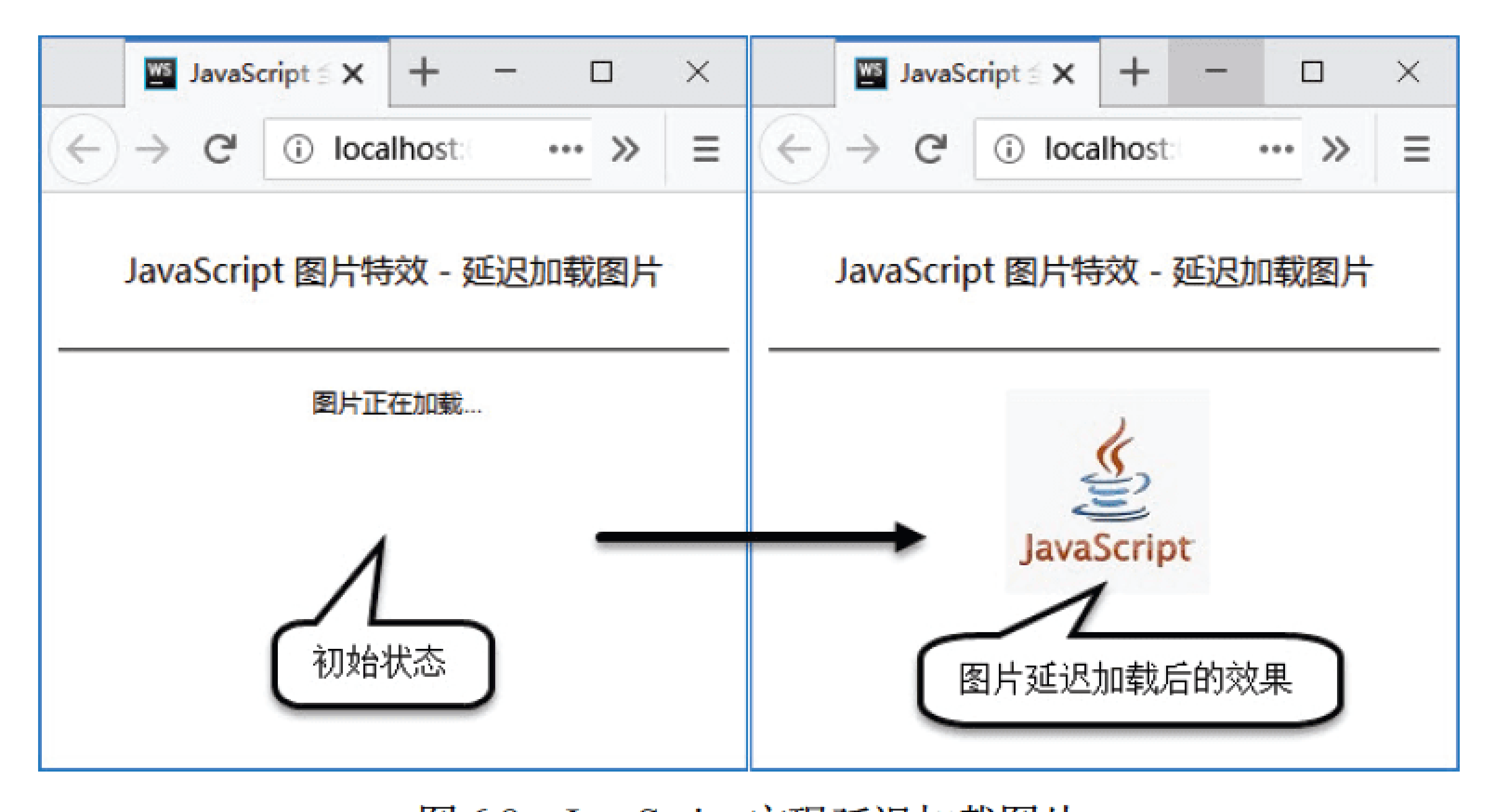


图 6.8　JavaScript 实现延迟加载图片

6.10 重新加载验证码图片

在 HTML 注册登录类网页设计中，图片验证码方式已经是一种主流技术解决方案了。使用图片验证码可以有效避免某些恶意注册的行为，有效提高网站使用的安全性。不过，由于图片验证码是通过纯 JavaScript 脚本语言算法模拟生成的，因此有时也会出现迷糊不清和字体过小的问题，此时就需要重新加载验证码图片，直到清晰可视为止。

下面看一个通过 JavaScript 模拟实现重新加载验证码图片的代码实例。

【代码 6-10】（详见源代码目录 ch06-js-img-reload-verify.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>重新加载验证码图片</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center">
    <input type="button" value="重新加载验证码图片"
           onclick="reloadVerityCode(this.id);"/>
    <img src="images/verifycode.01.jpg" alt="图片未加载" id="id-img"
         lazy_src_1="images/verifycode.01.jpg"
         lazy_src_2="images/verifycode.02.jpg"
         lazy_src_3="images/verifycode.03.jpg"
    />
</div>
</body>
<script type="text/javascript">
    var index = 1;
    /**
     * reload verify code image
     * @param thisid
     */
    function reloadVerityCode(thisid) {
       var img = document.getElementById('id-img');
       index++;
```

```
31          if(index > 3)
32              index = 1;
33          img.src = img.attributes['lazy_src_' + index.toString()].value;
34      }
35  </script>
36  </html>
```

关于【代码 6-10】的说明：

- 第 14 行代码通过<input>标签元素定义了一个按钮，并定义了单击 onclick 事件处理方法（reloadVerityCode()），用于测试“重新加载验证码图片”的操作。
- 第 15 ~ 19 行代码通过<img>标签元素定义的图片是关键所在。注意，第 16 ~ 18 行代码增加了一组自定义属性（lazy_src_1、lazy_src_2 和 lazy_src_3），其属性值引用了想要重新加载的图片验证码路径。
- 第 28 ~ 34 行代码是事件处理方法（reloadVerityCode()）的实现过程，通过图片对象的 attributes 属性获取自定义属性（lazy_src_1、lazy_src_2 和 lazy_src_3）的值，然后赋给 src 属性，从而实现重新加载验证码图片的操作。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 6.9 所示。

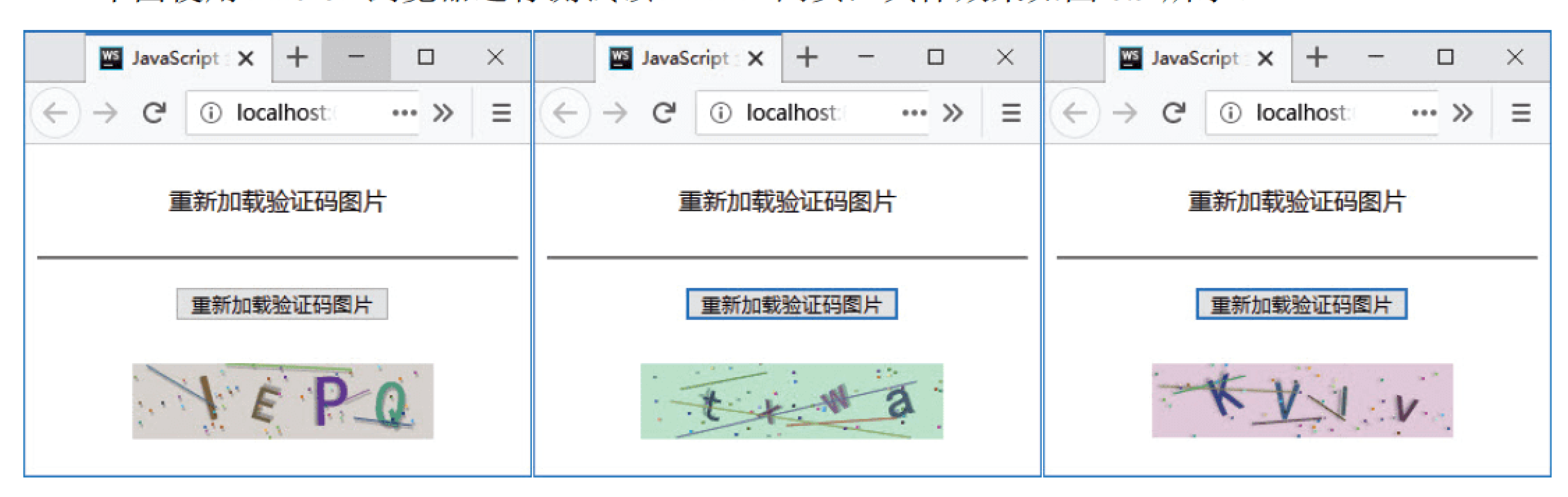


图 6.9　JavaScript 实现重新加载验证码图片

图 6.9 所示的 3 个页面演示了“重新加载验证码图片”操作的效果。

第 7 章　文本框和下拉列表框特效

本章介绍如何通过 JavaScript 来实现文本框和下拉列表框的各种特效，通过这些特效来丰富 HTML 表单的设计手段和展示效果。

7.1　文本框和下拉列表框概述

文本框和下拉列表框是 HTML 表单中使用频率很高的元素：文本框可以用来接受用户输入的信息；下拉列表框可以提供给用户若干个可供选择的选项数据，都是非常实用的表单控件。

在 HTML 表单中，通过使用<input>标签元素来定义文本框，通过使用<select>标签元素来定义下拉列表框。

文本框的语法格式如下：

【代码 7-1】

```
/** 文本框定义 */
<input  type="text|password|email…" />
```

其中，type 属性是必需的，用于指定文本框的类型，比如"text"表示普通的文本输入框、"password"表示密码输入框、"email"表示 Email 电子邮件格式输入框等。

下拉列表框的语法格式如下：

【代码 7-2】

```
/** 下拉列表框定义 */
<select>
  <option></option>
  <option></option>
  …
</select>
```

其中，<option>标签用于定义下拉列表框的选项。

通过 JavaScript 脚本语言可以为文本框和下拉列表框实现多种特效，比如只带下划线的文本框、默认大小写的文本框、动态添加和删除下拉选项、多级联动下拉列表框、可输入的下拉列表框、带格式校验的文本框等。后续将为读者介绍多种文本框和下拉列表框特效的代码实例。

7.2　只带下划线的文本框

在 HTML 表单中，文本框默认样式都是带有四条黑色方形边框的，在网页中这个样式是没有问题的。但是，如果将 HTML 表单打印出来的话，文本框的四条黑色方形边框样式就会显得很突兀，远没有 Word 文档中只带有下划线文本框样式的体验好。那么，HTML 表单能不能设计出只带下划线的文本框呢？其实，只要通过设定文本框 Style 对象的 border 属性就可以实现。

下面介绍一个通过 JavaScript 实现只带下划线的文本框的代码实例。

【代码 7-3】（详见源代码目录 ch07-js-input-underline.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>只带下划线的文本框</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center">
    只带下划线的文本框:  <input type="text" value=""
                       id="id-input-underline" />
</div>
</body>
<script type="text/javascript">
    /**
     * 初始化只带下划线的文本框
     * @param ev
     */
    window.onload = function (ev) {
        var inputUnderline = document.getElementById("id-input-underline");
        inputUnderline.style.borderColor = 'black';
                                            // TODO: 设置边框颜色
        inputUnderline.style.borderStyle = 'solid';
                                            // TODO: 设置边框样式为实线
        inputUnderline.style.borderWidth = '0 0 1px 0';
                                            // TODO: 设置边框大小，0 代表无
    };
```

```
28 </script>
29 </html>
```

关于【代码 7-3】的说明：

- 第 14 行代码通过<input>标签元素定义了一个文本框，用于实现“只带下划线的文本框”。
- 第 24 ~ 26 行代码通过分别设定文本框 Style 对象的边框颜色 borderColor、边框样式 borderStyle 和边框厚度 borderWidth 属性值实现了只带下划线的文本框样式。比较关键的是第 26 行代码定义的边框厚度 borderWidth 属性，“0 0 1px 0”数值组合分别表示“上、右、下、左”四条边框的厚度。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 7.1 所示。页面中的文本框（用户输入内容：1234567890）是只带有下划线样式的。

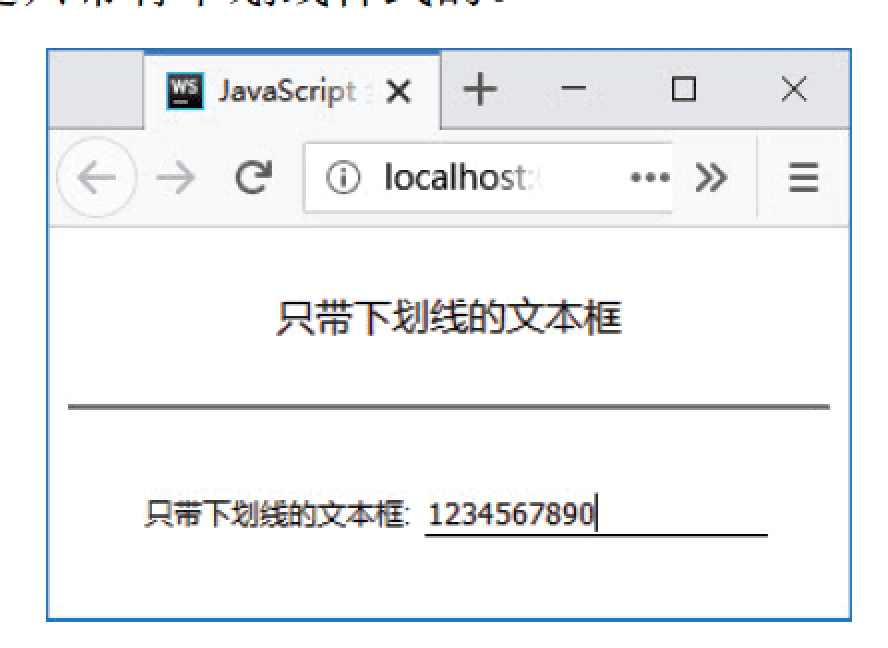


图 7.1　JavaScript 实现只带下划线的文本框

7.3　用正则表达式验证 Email 格式

随着 HTML 5 标准规范的推出，只要使用<input type="email">标签类型来定义文本框，就可以支持 Email 格式的验证了。在早期 HTML 标准规范中，如果想验证用户输入的 Email 格式是否正确，就需要使用 JavaScript 正则表达式功能来实现了。随着技术标准的不断进步，JavaScript 设计开发的门槛越来越低，早期需要手动编写代码来实现的功能，现在设计人员直接使用新的标准规范就可以完成了。不过，这并不等于不需要设计人员去了解掌握相关技术的实现原理，相信大多数读者是支持这个观点的。

下面介绍一个通过 JavaScript 实现用正则表达式验证 Email 格式的代码实例。

【代码 7-4】（详见源代码目录 ch07-js-input-regex-email.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
```

```
07 <body>
08 <!-- 添加文档主体内容 -->
09 <header>
10     <nav>用正则表达式验证 Email 格式</nav>
11 </header>
12 <!-- 添加文档主体内容 -->
13 <div id="id-div-center">
14     请输入 Email:<input type="text" id="id-input-email"
                        onblur="validateEmail(this.id)"/>
15     验证结果:<input type="text" id="id-input-result" readonly>
16 </div>
17 </body>
18 <script type="text/javascript">
19     /**
20      * validate email format
21      * @param thisid
22      */
23     function validateEmail(thisid) {
24         var inputResult = document.getElementById("id-input-result");
25         var inputEmail = document.getElementById(thisid);
                                            // TODO:获得文本框的 DOM
26         var email = inputEmail.value;   // TODO: 获得用户输入的 Email
27         // TODO: 定义正则表达式
28         var regexEmail = /^([a-zA-Z0-9_-])+@([a-zA-Z0-9_-])+
                                            ((\.[a-zA-Z0-9_-]{2,3}){1,2})$/;
29         if (regexEmail.test(email)) {
30             inputResult.value = "Email 校验通过.";
31         } else {
32             inputResult.value = "Email 校验失败,请检查输入格式.";
33         }
34     }
35 </script>
36 </html>
```

关于【代码 7-4】的说明：

- 第 14 行代码通过<input id="id-input-email">标签元素定义了一个文本框，并定义了 onblur 事件处理方法（validateEmail()），用于测试“用正则表达式验证 Email 格式”的过程。
- 第 15 行代码通过<input>标签元素定义了另一个文本框（只读 readonly 类型），用于显示验证 Email 格式的结果。

- 第 23～34 行代码是 validateEmail()方法的实现过程，关键是第 28 行代码定义的正则表达式（regexEmail）；第 29～33 行代码通过调用正则表达式（regexEmail）对象的 test()方法来验证 Email 格式是否正确，然后将验证结果显示到第 15 行代码定义的只读文本框中。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 7.2 所示。

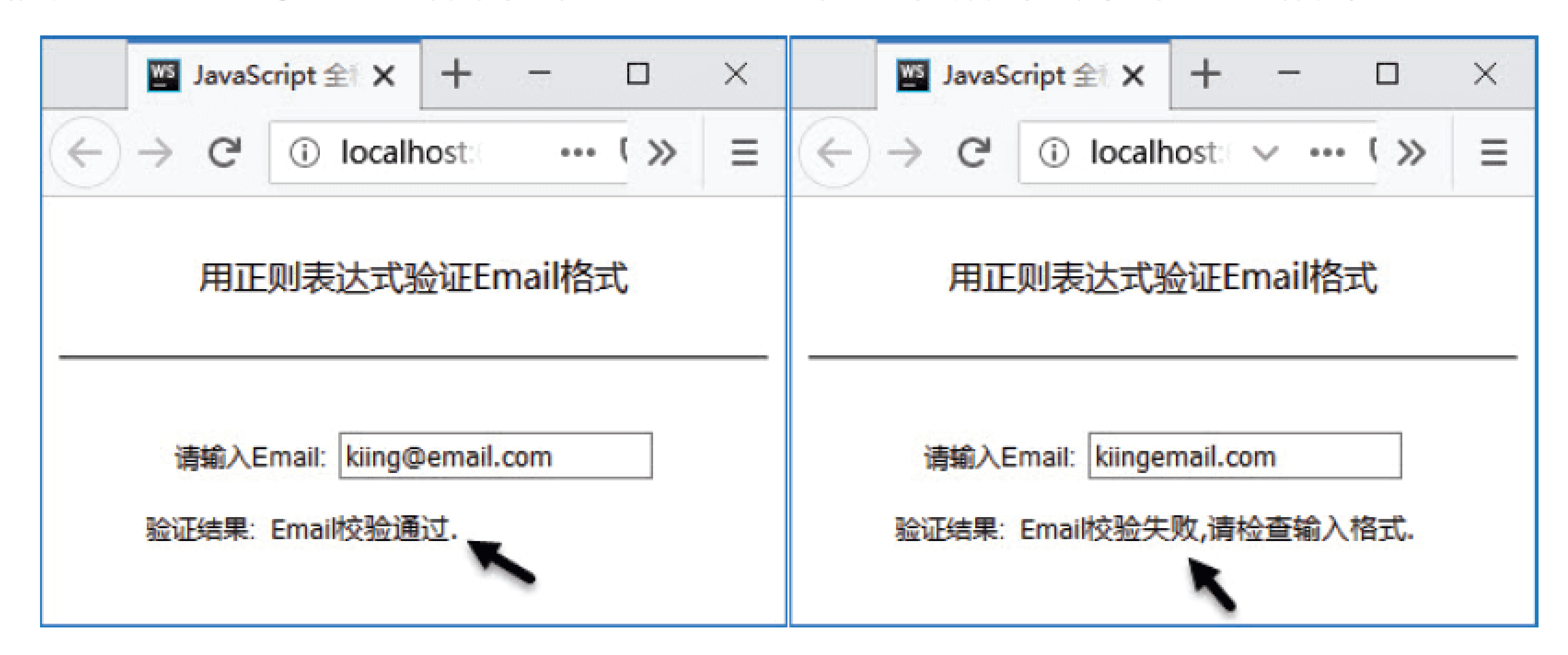


图 7.2　JavaScript 实现用正则表达式验证 Email 格式

如图 7.2 中箭头所示，在文本框中输入正确的电子邮件地址（king@email.com）后，提示“Email 校验通过”了。而再将输入内容中的“@”字符删除后，提示“Email 校验失败，请检查输入格式”。

7.4　首字母或全部字母大写

在一些文本框的设计场景下，需要将用户输入文本的首字母或全部字母自动强制转换为大写格式，JavaScript 语法中的 String 对象提供了一个 toUpperCase()方法来进行字母大写转换。下面就看一个通过 JavaScript 实现首字母或全部字母大写的代码实例。

【代码 7-5】（详见源代码目录 ch07-js-input-alphabet-upper.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>首字母或全部字母大写</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center">
    全部字母大写:<input type="text" id="id-input-first"
                     onblur="allAlphabetUpper(this.id);" />
```

```
15     首字母大写:<input type="text" id="id-input-all"
                        onblur="firstAlphabetUpper(this.id);" />
16  </div>
17  </body>
18  <script type="text/javascript">
19     /**
20      * Set all alphabet to upper case
21      * @param thisid
22      */
23     function allAlphabetUpper(thisid) {
24        var allAlpha = document.getElementById(thisid);
25        var valAll = allAlpha.value;
26        var valNewAll = valAll.toUpperCase();
27        allAlpha.value = valNewAll;
28     }
29     /**
30      * Set first alphabet to upper case
31      * @param thisid
32      */
33     function firstAlphabetUpper(thisid) {
34        var firstAlpha = document.getElementById(thisid);
35        var valFirst = firstAlpha.value;
36        var firstUpper = valFirst.charAt(0).toUpperCase();
37        var valNewFirst = firstUpper + valFirst.substr(1, valFirst.length);
38        firstAlpha.value = valNewFirst;
39     }
40  </script>
41  </html>
```

关于【代码 7-5】的说明：

- 第 14 行代码通过<input id="id-input-first">标签元素定义了第一个文本框，并定义了 onblur 事件处理方法（allAlphabetUpper()），用于测试"全部字母转换为大写"的方法。
- 第 15 行代码通过<input id="id-input-all">标签元素定义了第二个文本框，并定义了 onblur 事件处理方法（firstAlphabetUpper()），用于测试"首字母转换为大写"的方法。
- 第 23～28 行代码是 allAlphabetUpper()方法的实现过程，关键是第 26 行代码通过使用 toUpperCase()方法实现将全部字母转换为大写的操作。
- 第 33～39 行代码是 firstAlphabetUpper()方法的实现过程，关键是第 36 行代码先通过使用 charAt()方法获取首字母，再通过使用 toUpperCase()方法实现将首字母转换为大写的操作。另外，第 37 行代码通过使用 substr()方法获取了除去首字母之后的全部字母，然后将大写首字母与其连接到一起。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 7.3 和图 7.4 所示。

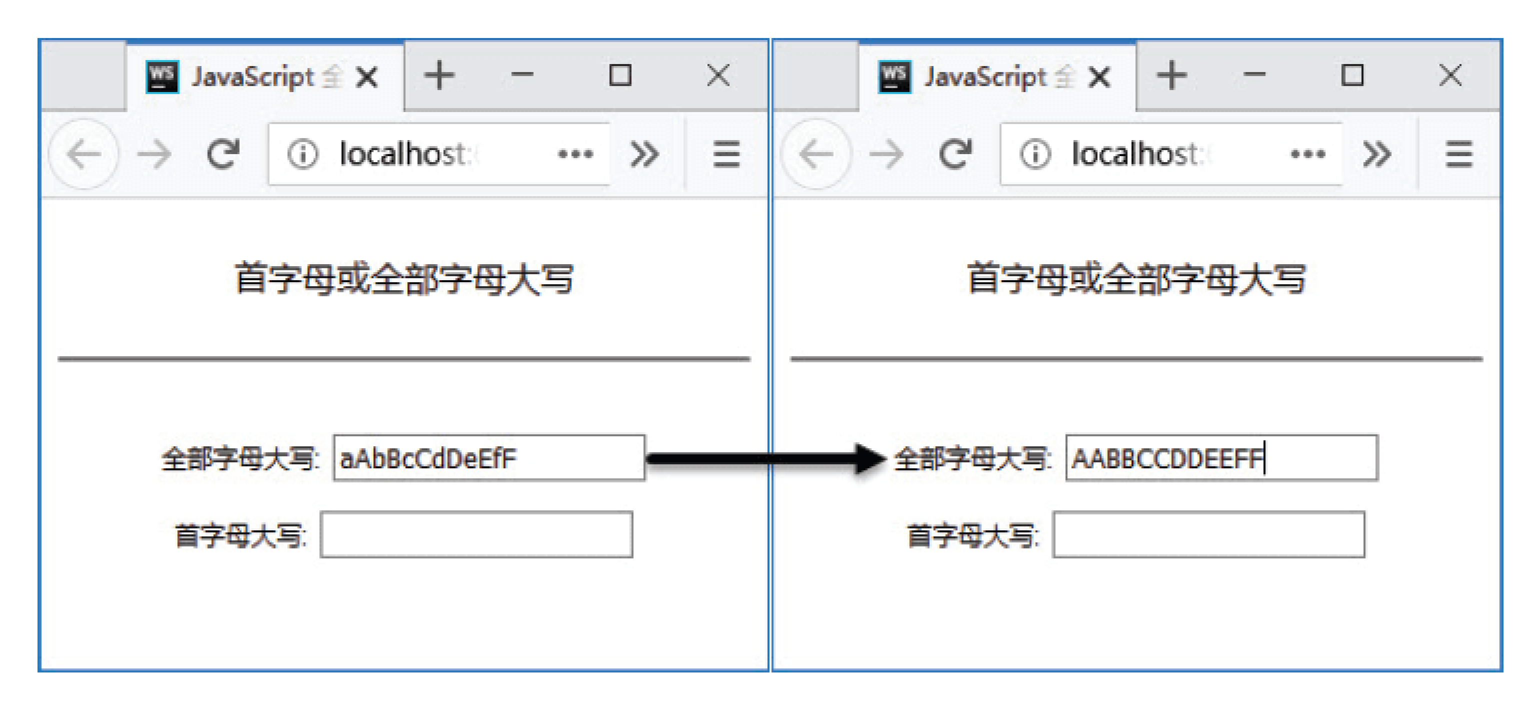


图 7.3　JavaScript 实现全部字母大写

如图 7.3 中箭头所示，页面中显示了“全部字母大写”的效果。

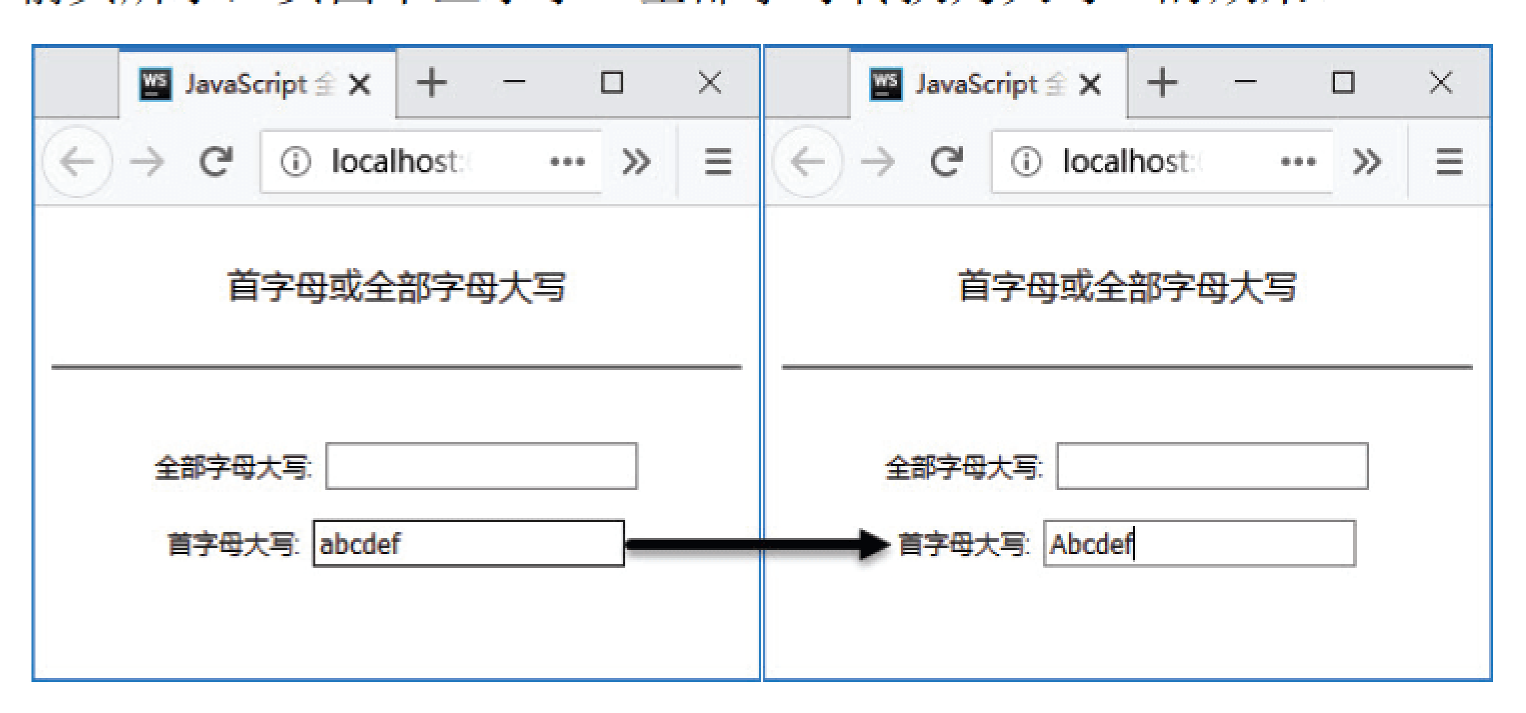


图 7.4　JavaScript 实现首字母大写

如图 7.4 中箭头所示，页面中显示了“首字母转换为大写”的效果。

7.5　只能输入数字的文本框

在一些文本框的设计场景下，可能只接受用户的输入文本为纯数字格式，那么通过正则表达式就可以实现对数字格式的验证。下面就看一个通过 JavaScript 实现只能输入数字的文本框的代码实例。

【代码 7-6】（详见源代码目录 ch07-js-input-regex-number.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
```

```
10      <nav>只能输入数字的文本框</nav>
11  </header>
12  <!-- 添加文档主体内容 -->
13  <div id="id-div-center">
14      只能输入数字:<input type="text" id="id-input-number"
                                        onblur="validateNumber(this.id)"/>
15      验证结果:<input type="text" id="id-input-result" readonly>
16  </div>
17  </body>
18  <script type="text/javascript">
19      /**
20       * validate number format
21       * @param thisid
22       */
23      function validateNumber(thisid) {
24          var inputResult = document.getElementById("id-input-result");
25          var inputNum = document.getElementById(thisid);
26          var valNum = inputNum.value;
27          var regexNum = /\b[0-9]+\b/;
28          if (regexNum.test(valNum)) {
29              inputResult.value = "数字格式验证通过.";
30          } else {
31              inputResult.value = "数字格式验证未通过,只能输入数字.";
32          }
33      }
34  </script>
35  </html>
```

关于【代码 7-6】的说明：

- 第 14 行代码通过<input id="id-input-number">标签元素定义了一个文本框，并定义了 onblur 事件处理方法（validateNumber()），用于测试“只能输入数字的文本框”的功能。
- 第 15 行代码通过<input>标签元素定义了另一个文本框（只读 readonly 类型），用于显示验证数字格式的结果。
- 第 23～33 行代码是 validateNumber()方法的实现过程，关键是第 27 行代码定义的正则表达式（regexNum）；第 28～32 行代码通过调用正则表达式（regexNum）对象的 test()方法来验证数字格式是否正确，然后将验证结果显示到第 15 行代码定义的只读文本框中。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 7.5 所示。

如图 7.5 中箭头所示，在文本框中输入文本（1234567890）后，提示“数字格式验证通过”；在文本框中输入文本（1a2b3c）后，因为是非纯数字，所以提示“数字格式验证未通过，只能输入数字”。

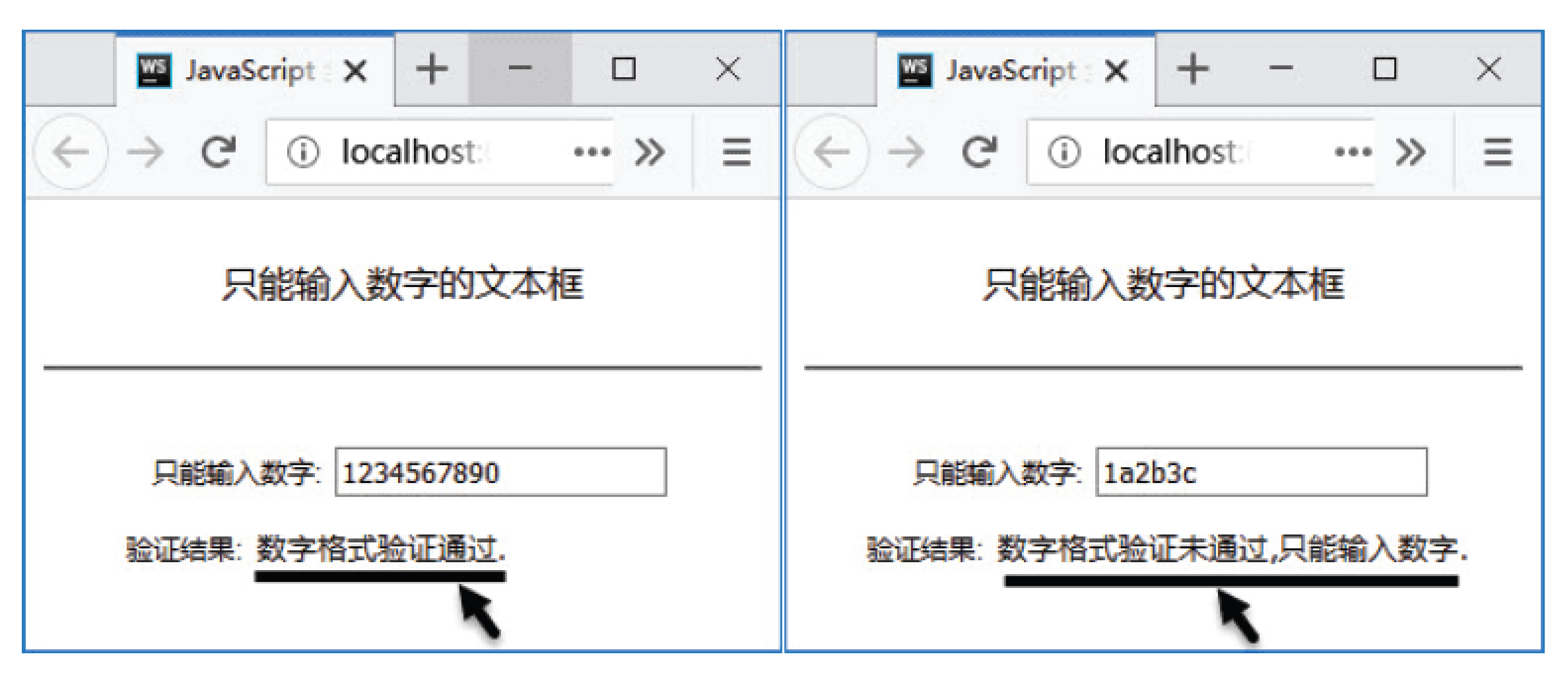


图 7.5　JavaScript 实现只能输入数字的文本框

7.6　判断字符的个数

在网站“留言板”或“发表评论”的页面中，读者一定见过类似提示信息“还可以继续输入 xx 个字符”的功能设计，这里就用到了判断用户输入字符个数的算法。下面就看一个通过 JavaScript 实现判断字符（支持区分中文字符）个数的代码实例。

【代码 7-7】（详见源代码目录 ch07-js-input-count-char.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 文本框特效 - 判断字符的个数</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center">
    请输入:<input type="text" id="id-input-count-char"
                              onblur="countChar(this.id)"/>
    输出:<input type="text" id="id-input-result" readonly>
</div>
</body>
<script type="text/javascript">
    /**
     * 判断字符个数
     * @param thisid
     */
```

```
23      function countChar(thisid) {
24          var vResult = document.getElementById("id-input-result");
25          //获得文本框的 DOM
26          var vInput = document.getElementById(thisid);
27          var vText = vInput.value;
28          var sumTotal = 0, sumEn = 0, sumCh = 0;
29          sumTotal = vText.length;
30          for (var i = 0; i < sumTotal; i++) {
31              var c = vText.charAt(i);
32              if (isChineseChar(c)) {
33                  sumCh++;
34              } else {
35                  sumEn++;
36              }
37          }
38          vResult.value = "共计" + sumTotal + "个字符,汉字字符为" + sumCh +
                                "个,非汉字字符为" + sumEn + "个.";
39      }
40      /**
41       * 判断汉字字符个数
42       * @param c
43       * @returns {boolean}
44       */
45      function isChineseChar(c) {
46          var reg = /[\u4E00-\u9FA5]/;    // TODO: 定义正则表达式
47          return reg.test(c);
48      }
49  </script>
50  </html>
```

关于【代码 7-7】的说明：

- 第 14 行代码通过<input id="id-input-count-char">标签元素定义了一个文本框，并定义了 onblur 事件处理方法（countChar()），用于测试“判断字符的个数”的功能。
- 第 15 行代码通过<input>标签元素定义了另一个文本框（只读 readonly 类型），用于显示判断字符个数的结果。
- 第 23 ~ 39 行代码是 countChar()方法的实现过程，关键是第 32 行代码调用的 isChineseChar()方法，该方法用于判断字符是否为汉字字符。
- 第 45 ~ 48 行代码是 isChineseChar()方法的实现过程。其中，第 46 行代码定义的正则表达式可用于判断汉字字符，第 47 行代码通过调用 test()方法返回判断结果。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 7.6 所示。

如图 7.6 中箭头所示，提示信息中显示了总计字符个数、汉字字符个数和非汉字字符个数。

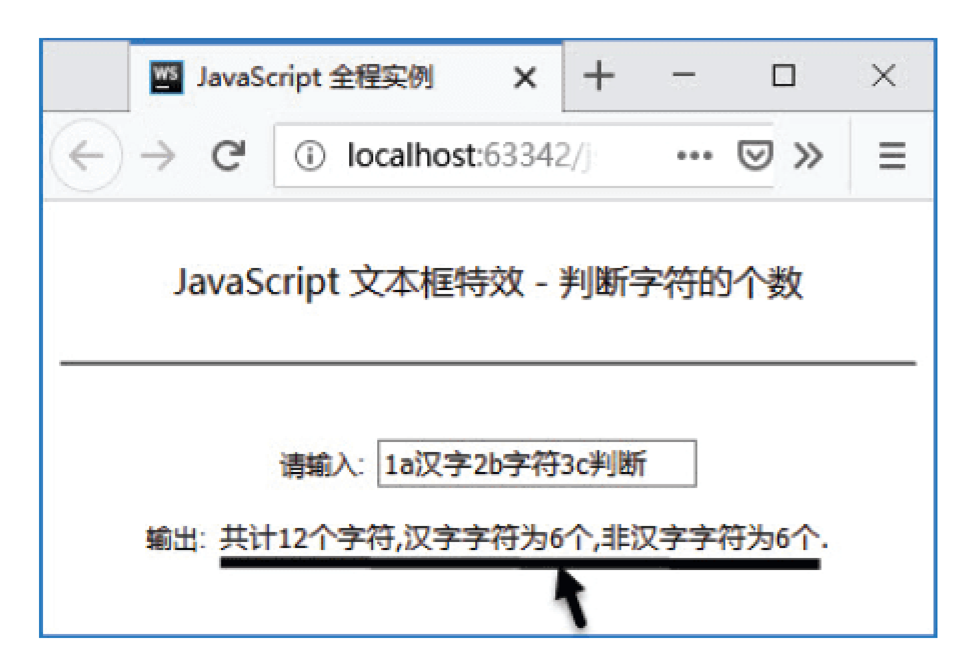


图 7.6　JavaScript 实现判断字符的个数功能

7.7　文本框获取焦点后自动清除内容

在用户需要重新输入文本框内容的场景下，往往用户要手动清除文本框中的原始内容，操作起来不太方便。于是，设计人员就想到了利用文本框获取焦点 onfocus()事件来自动清除文本框中的原始内容。下面就看一个通过 JavaScript 实现文本框获取焦点后自动清除内容的代码实例。

【代码 7-8】（详见源代码目录 ch07-js-input-focus-clear.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>文本框获取焦点后自动清除内容</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center">
    文本框:  
    <input type="text"
           value="文本框获取焦点后自动清除内容"
           id="id-input-focus-clear"
           onfocus="focusClearContent(this.id)"/><br/>
</div>
</body>
<script type="text/javascript">
    /**
     * on input get focus, clear it's content
```

```
24        * @param thisid
25        */
26       function focusClearContent(thisid) {
27           var input = document.getElementById(thisid);
28           input.value = '';  // TODO: 把文本内容的值设为空字符
29       }
30   </script>
31   </html>
```

关于【代码 7-8】的说明：

- 第 15～18 行代码通过<input id="id-input-focus-clear">标签元素定义了一个文本框，并定义了 onfocus 事件处理方法（focusClearContent()），用于测试“文本框获取焦点后自动清除内容”的操作。
- 第 26～29 行代码是 focusClearContent()方法的实现过程，关键是第 28 行代码，通过将文本框的 value 属性值清空来实现文本框在获取焦点后自动清除文本框内容的功能。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 7.7 所示。

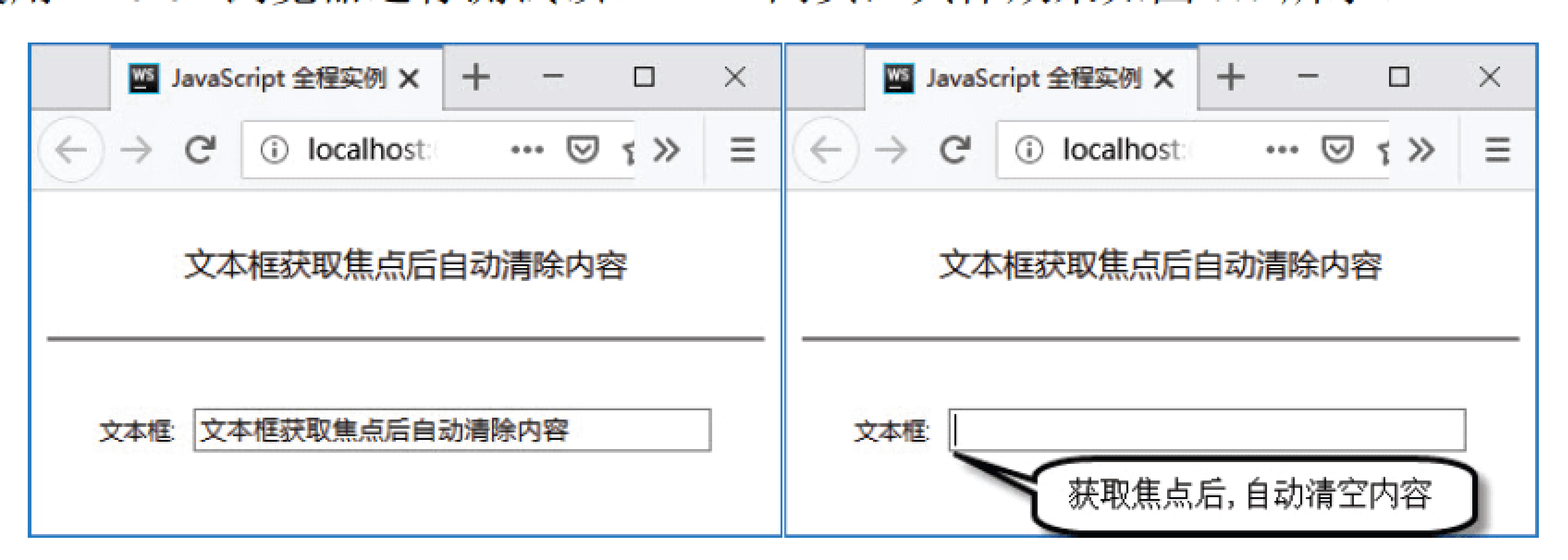


图 7.7　JavaScript 实现文本框获取焦点后自动清除内容

7.8　清空所有文本型输入框

用户在操作表单时，有时需要清空所有文本型输入框的内容，这些文本框不但包括“text”类型，还包括“password”或“email”类型等。此时，就需要设计人员先根据<input>标签元素的 type 属性来判断再有针对性地进行操作。下面看一个通过 JavaScript 实现清空所有文本型输入框的代码实例。

【代码 7-9】（详见源代码目录 ch07-js-input-all-clear.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
```

```
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>清空所有文本型输入框</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center">
    文本框:  <input type="text" value="输入框text类型"/><br/><br/>
    文本框:  <input type="password" value="123456"/><br/><br/>
    文本框:  <input type="email" value="king@email.com"/>
                                                                <br/><br/>
    <button onclick="clearAllText()">清空输入框(text)</button><br/>
    <button onclick="clearAllInput()">清空输入框(text|password|email)
                                                                </button><br/>
</div>
</body>
<script type="text/javascript">
    /**
     * clear input, type = text
     */
    function clearAllText() {
        var inputs = document.getElementsByTagName("input");
        for(let i=0; i<inputs.length; i++) {
            if(inputs[i].type == "text") {
                inputs[i].value = "";
            }
        }
    }
    /**
     * clear input, type = text|password|email
     */
    function clearAllInput() {
        var inputs = document.getElementsByTagName("input");
        for(let i=0; i<inputs.length; i++) {
            if(inputs[i].type == "text"||inputs[i].type==
                                  "password"||inputs[i].type == "email") {
                inputs[i].value = "";
            }
        }
    }
```

```
44  </script>
45  </html>
```

关于【代码 7-9】的说明：

- 第 14～16 行代码通过<input>标签元素定义了一组文本框，且其 type 属性值分别为 text 类型、password 类型和 email 类型，用于测试“清空所有文本型输入框”的操作。
- 第 17 行代码通过<button>标签元素定义了第一个按钮，并定义了单击 onclick 事件处理方法（clearAllText()），用于进行清除全部 text 类型文本框中内容的操作。
- 第 18 行代码通过<button>标签元素定义了第二个按钮，并定义了单击 onclick 事件处理方法（clearAllInput()），用于进行清除全部 text 类型、password 类型和 email 类型文本框中内容的操作。
- 第 25～32 行代码是事件处理方法（clearAllText()）的实现过程，关键是第 28 行代码，先通过判断文本框的 type 属性值是否为 text 类型再进行清空所有文本型输入框的操作。
- 第 36～43 行代码是事件处理方法（clearAllInput()）的实现过程，与 clearAllText()方法的区别就是第 39 行代码，先通过判断文本框的 type 属性值是否为 text 类型、password 类型和 email 类型中的任一类，再进行清空所有文本型输入框的操作。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 7.8 所示。

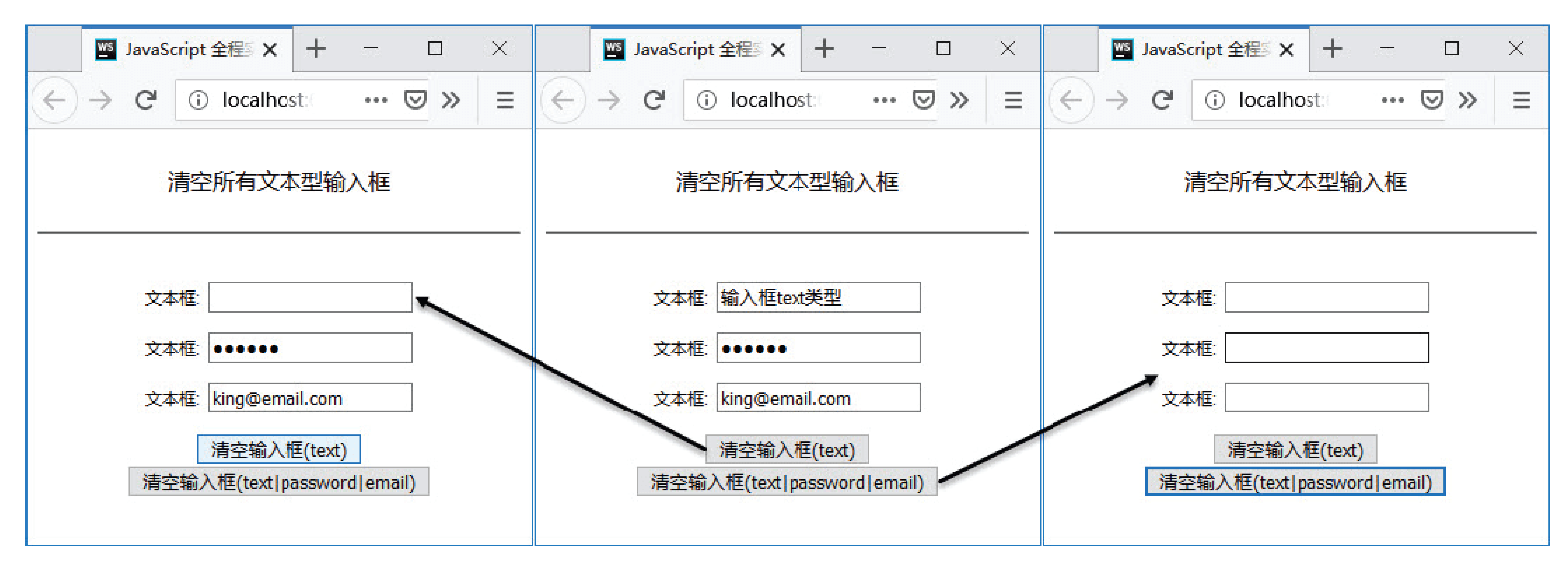


图 7.8　JavaScript 实现清空所有文本型输入框

如图 7.8 中箭头所示，中间的页面为原始状态，左侧页面为“清空文本框(text)”内容后的效果，右侧页面为“清空文本框(text|password|email)”内容后的效果。

7.9　校验电话号码格式

在 HTML 页面表单中，校验用户录入的电话号码格式是否正确是一项很常用的设计功能。本节主要以国内常用的固定电话和移动电话号码为样本目标，通过 JavaScript 正则表达式来测试完成校验电话号码格式的操作。下面看一个通过 JavaScript 实现校验电话号码格式的代码实例。

【代码 7-10】（详见源代码目录 ch07-js-input-regex-tel.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 文本框特效 - 校验电话号码格式</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center">
    电话号码:<input type="text" id="id-input-tel"
                              onblur="validateTel(this.id)"/>
    验证结果:<input type="text" id="id-input-result" readonly>
</div>
</body>
<script type="text/javascript">
    function validateTel(thisid) {
        var vResult = document.getElementById("id-input-result");
        var vTel = document.getElementById(thisid);
        var vTelVal = vTel.value;
        var regexTel = /^[(0-9)-]+$/;
        var regexMobile = /^1[3578]\d{9}$/;
        var regexTel1 = /^0\d{2}-\d{8}$|^0\d{3}-\d{7}$/;
        var regexTel2 = /^\(0\d{2}\)\d{8}$|^\(0\d{3}\)\d{7}$/;
        if (regexTel.test(vTelVal)) {
            if (regexMobile.test(vTelVal)) {
                vResult.value = "正确的手机号码格式.";
            } else if (regexTel1.test(vTelVal) || regexTel2.test(vTelVal)){
                vResult.value = "正确的固定电话号码格式.";
            } else {
                vResult.value = "电话号码格式不正确,请重新检查.";
            }
        } else {
            vResult.value = "电话号码格式不正确,请重新检查.";
        }
    }
</script>
</html>
```

关于【代码 7-10】的说明：

- 第 14 行代码通过<input id="id-input-tel">标签元素定义了一个文本框，并定义了 onblur 事件处理方法（validateTel()），用于测试“校验电话号码格式”的操作。
- 第 15 行代码通过<input>标签元素定义了另一个文本框（只读 readonly 类型），用于校验电话号码格式的结果。
- 第 19 ~ 38 行代码是 validateTel()方法的实现过程，关键是第 23 ~ 26 行代码定义的一组正则表达式，分别用来验证手机号码和固定电话号码。
    - 第 23 行代码定义的正则表达式（regexTel）是对用户输入电话号码的一个大体测试。因为国内手机号码只包括数字（不考虑国际区号），国内固定电话的区号和电话号码之间一般通过短横杠“-”或小括号“( )”来连接。因此，正则表达式（regexTel）只对数字和短横杠“-”或小括号“( )”来进行验证。
    - 第 24 行代码定义的正则表达式（regexMobile）是对用户输入手机号码的一个较为精确的测试。国内手机号码是固定的 11 位数字（不考虑国际区号），且目前的首位号码固定为数字 1，第二位号码为固定数字 3、5、7 和 8。
    - 第 25 ~ 26 行代码定义的正则表达式（regexTel1 和 regexTel2）是对用户输入固定电话号码的一个较为精确的测试。正则表达式（regexTel1）针对的是使用短横杠“-”符号连接的固定电话，而正则表达式（regexTel2）针对的是使用小括号“( )”符号连接的固定电话。

下面使用 Firefox 浏览器运行测试该 HTML 网页，针对手机号码的具体效果如图 7.9 所示。

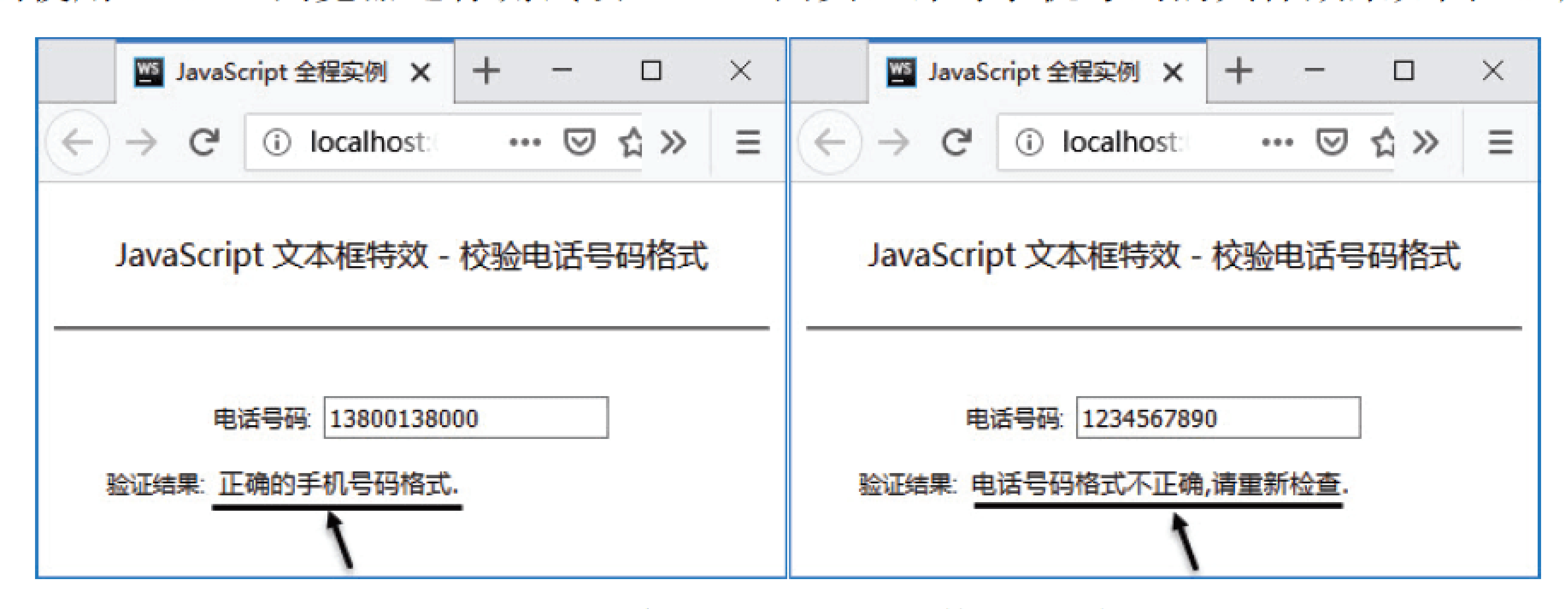


图 7.9　JavaScript 实现校验电话号码格式（手机号码）

针对固定电话号码的具体效果如图 7.10 所示。

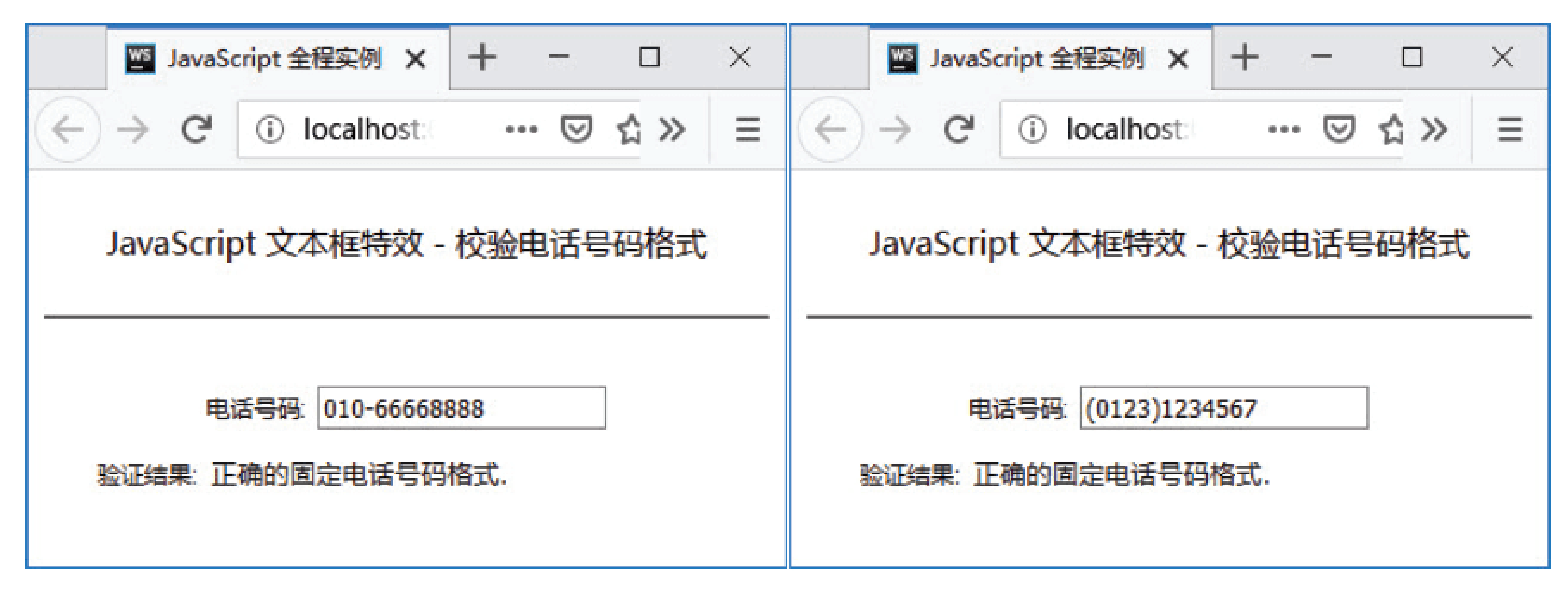


图 7.10　JavaScript 实现校验电话号码格式（固定电话号码）

7.10 鼠标划过文本框改变其背景色

如果想突出显示 HTML 表单中比较重要的文本框，那么当用户鼠标划过该文本框时改变其背景颜色是一个不错的选择。下面看一个通过 JavaScript 实现鼠标划过文本框改变其背景色的代码实例。

【代码 7-11】（详见源代码目录 ch07-js-input-mouseover-bgcolor.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>鼠标划过文本框改变其背景色</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center">
    <!-- 定义文本框 -->
    文本框:  <input type="text"/>
    文本框:  <input type="text"/>
    文本框:  <input type="text"/>
</div>
</body>
<script type="text/javascript">
    /**
     * window on load event
     * @param ev
     */
    window.onload = function (ev) {
        var inputs = document.getElementsByTagName('input');
        // TODO: 遍历所有文本框
        for (var i = 0; i < inputs.length; i++) {
            var input = inputs[i];  // TODO: 当前文本框
            input.onmouseover = function () {
                this.style.backgroundColor = 'red'; // TODO: 修改背景色为红色
            };
            input.onmouseout = function () {
                this.style.backgroundColor = '';    // TODO: 恢复背景色
```

```
35              };
36          }
37      };
38  </script>
39  </html>
```

关于【代码 7-11】的说明：

- 第 15 ~ 17 行代码通过<input type="text">标签元素定义了一组文本框，用于实现“鼠标划过文本框改变其背景色”的功能。
- 第 28 ~ 36 行代码通过 for 循环语句遍历全部文本框，并依次为每一个文本框添加鼠标经过 onmouseover 事件处理方法和鼠标移出 onmouseout 事件处理方法，在事件处理方法中相应地为文本框添加和取消背景颜色。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 7.11 所示。

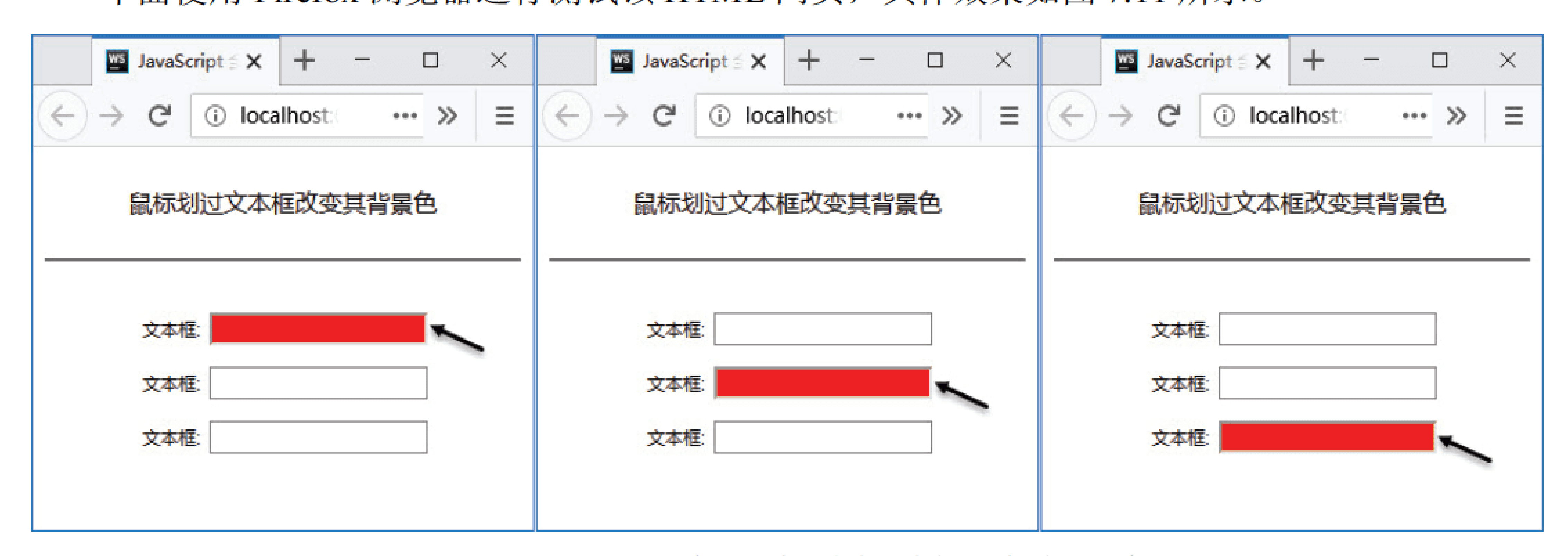


图 7.11　JavaScript 实现鼠标划过文本框改变其背景色

如图 7.11 中箭头所示，页面中分别演示了鼠标依次划过每个文本框后改变其背景颜色的效果。

7.11　设置下拉列表框的值

在 HTML 页面中，用户可以通过下拉列表框手动选取自己需要的选项值。不通过手动选择，能不能通过程序设置下拉列表框的值呢？答案是完全可以的，而且通过 JavaScript 脚本语言设置下拉列表框的值是一项非常有用的功能。下面看一个通过 JavaScript 实现设置下拉列表框的值的代码实例。

【代码 7-12】（详见源代码目录 ch07-js-select-set-value.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
```

```
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>设置下拉菜单的值</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center">
    <!-- 定义下拉列表框 -->
    设置颜色:  
    <select name="selSetColor" id="idSelSetColor">
        <option>请选择...</option>
        <option value="red">Red</option>
        <option value="yellow">Yellow</option>
        <option value="green">Green</option>
    </select>
    <!-- 定义 Radio 单选 -->
    请选择颜色:  
    <input name="radioColor" value="red"
            onclick="setSelValue(this.value);" checked/>Red
    <input name="radioColor" value="yellow"
            onclick="setSelValue(this.value);"/>Yellow
    <input name="radioColor" value="green"
            onclick="setSelValue(this.value);"/>Green
</div>
</body>
<script type="text/javascript">
    window.onload = function (ev) {
        var idSel = document.getElementById('idSelSetColor');
        var arrRadio = document.getElementsByName("radioColor");
        for (let i = 0; i < arrRadio.length; i++) {
            if (arrRadio[i].checked) {
                idSel.value = arrRadio[i].value;
            }
        }
    };
    /**
     * set select value
     * @param val
     */
    function setSelValue(val) {
        var idSel = document.getElementById('idSelSetColor');
        idSel.value = val;
```

```
46       }
47   </script>
48   </html>
```

关于【代码 7-12】的说明：

- 第 16～21 行代码通过<select id="idSelSetColor">标签元素定义了一个下拉列表框，并定义了一组颜色选项，用于实现“设置下拉列表框的值”的操作。
- 第 24～26 行代码通过<input type="radio">标签元素定义了一组单选框，并分别定义了单击 onclick 事件处理方法（setSelValue()），用于设置上面定义的下拉列表框的颜色值。其中，第 24 行代码定义的单选框（Red）通过添加“checked”属性，被定义为默认选中项。
- 第 30～38 行代码定义了页面加载 onload 事件处理方法，首先判断哪一项单选框是被选中（checked 属性值为 true）的，然后将该单选框的颜色值设置为下拉列表框的初始化颜色值选项。
- 第 43～46 行代码是 setSelValue()事件处理方法的实现过程，通过将传递进来的单选框颜色值赋给下拉列表框，实现通过 JavaScript 脚本语言“设置下拉列表框的值”的操作。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 7.12 所示。

如图 7.12 中箭头所示，当用户通过单选框选择颜色后，下拉列表框的颜色选项也被同步设置为对应的颜色了。

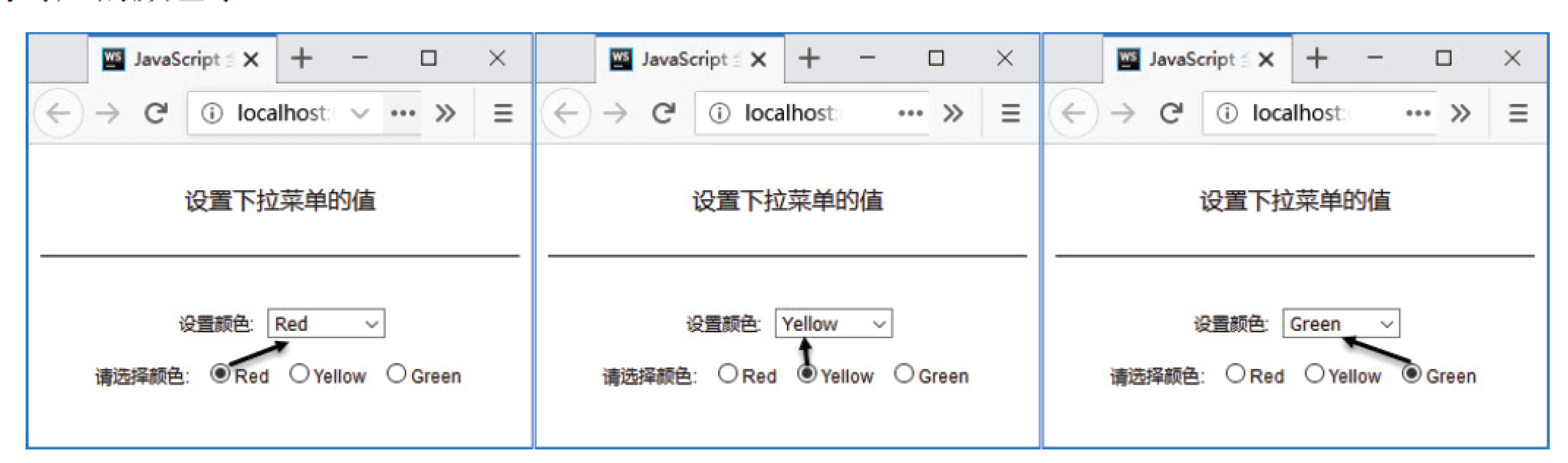


图 7.12　JavaScript 实现设置下拉列表框的值

7.12　动态添加下拉列表框选项

7.11 节中介绍了设置下拉列表框的值，本节继续介绍如何动态添加下拉列表框的选项。在 HTML DOM 对象中，下拉列表框 Select 对象中定义有一个 options 集合，代表着下拉列表框中全部选项的集合。通过调用 options 集合的 add(option[,])方法，就可以为 Select 对象添加一个选项。下面看一个通过 JavaScript 实现动态添加下拉列表框选项的代码实例。

【代码 7-13】（详见源代码目录 ch07-js-select-add-option.html 文件）

```
<!doctype html>
<html lang="en">
<head>
```

```
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>动态添加下拉列表框选项</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center">
    <table>
        <tr>
            <th>设置颜色:  </th>
            <td>
                <!-- 定义下拉列表框 -->
                <select name="selSetColor" id="idSelSetColor">
                    <option>请选择...</option>
                    <option value="red">Red</option>
                    <option value="yellow">Yellow</option>
                    <option value="green">Green</option>
                </select>  
            </td>
            <th>添加颜色:  </th>
  <td><input type="button" id="id-add-color" value="添加颜色"
               onclick="addColor()"/></td>
        </tr>
        <tr>
            <td></td>
            <td></td>
            <td>value:  </td>
            <td><input type="text" id="id-sel-val" value=""/></td>
        </tr>
        <tr>
            <td></td>
            <td></td>
            <td>text:  </td>
            <td><input type="text" id="id-sel-text" value=""/></td>
        </tr>
    </table>
</div>
</body>
<script type="text/javascript">
    /**
```

```
46        * add option color
47        * @param thisid
48        */
49       function addColor(thisid) {
50          var idSel = document.getElementById('idSelSetColor');
51          var idSelVal = document.getElementById('id-sel-val');
52          var idSelText = document.getElementById('id-sel-text');
53          var val = idSelVal.value;
54          var text = idSelText.value;
55          idSel.options.add(new Option(val, text));
56       }
57   </script>
58   </html>
```

关于【代码 7-13】的说明：

- 第 19 ~ 24 行代码通过<select id="idSelSetColor">标签元素定义了一个下拉列表框，并初始化了一组颜色选项，用于测试“动态添加下拉列表框选项”的操作。
- 第 27 行代码通过<input type="button">标签元素定义了一个按钮，并定义了单击 onclick 事件方法（addColor()），用于将用户输入的选项值（见第 33 行和第 39 行代码的解释）添加进下拉列表框。
- 第 33 行和第 39 行代码分别通过<input type="text">标签元素定义了一组文本框，用于用户输入下拉列表框选项<option>的 value 属性值和 text 文本内容。
- 第 49 ~ 56 行代码是 addColor()事件处理方法的实现过程，首先获取用户输入的选项<option>值，然后在第 55 行代码通过 options 集合的 add()方法将 Option 对象添加进下拉列表框。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 7.13 所示。

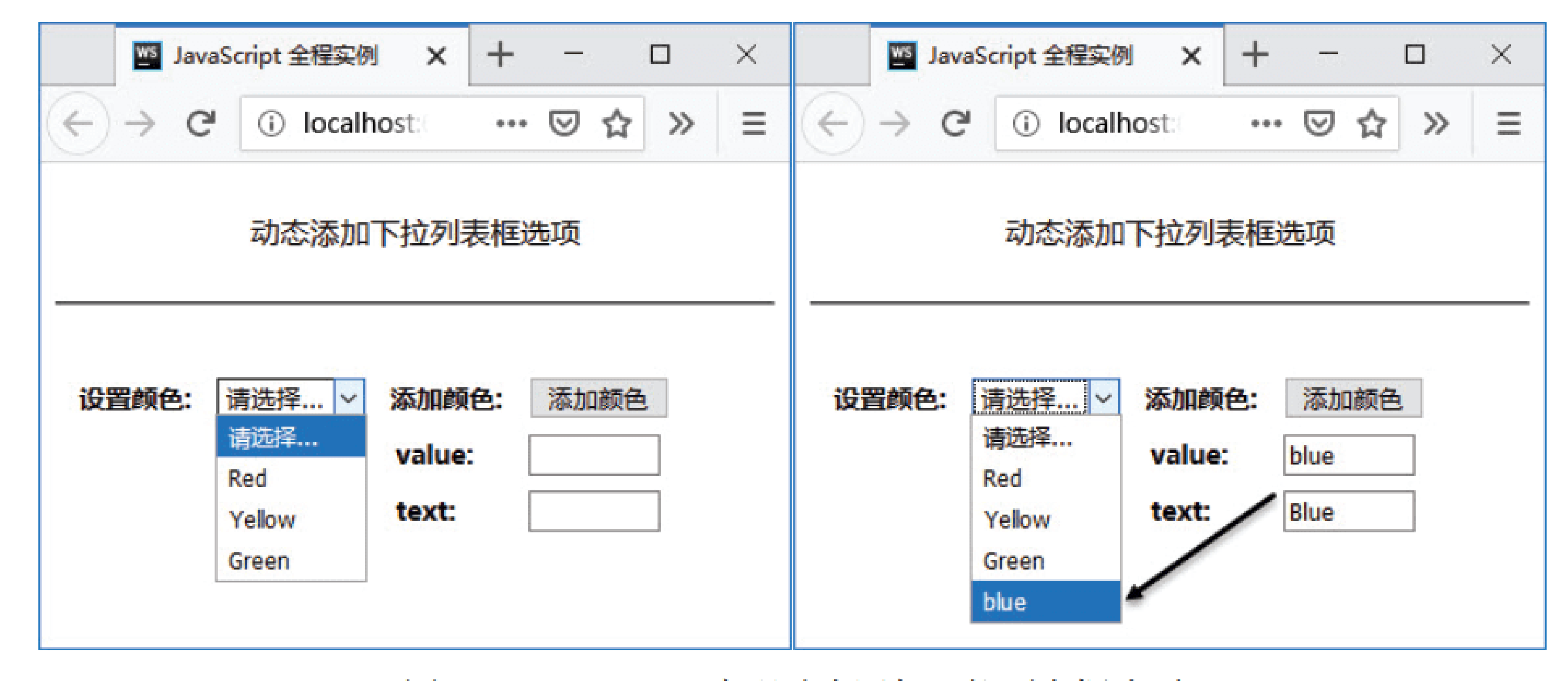


图 7.13　JavaScript 实现动态添加下拉列表框选项

如图 7.13 中箭头所示，通过“添加颜色”按钮成功将颜色（blue）添加进下拉列表框中。

7.13 动态删除下拉列表框选项

本节接着前文的内容，继续介绍如何动态删除下拉列表框的选项。在 HTML DOM 对象中，通过调用下拉列表框 options 集合的 remove(selectedIndex)方法，就可以为 Select 对象删除一个选项。通过将 options 集合的长度（length）属性设置为零，就可以将 Select 对象的全部选项进行删除。下面看一个通过 JavaScript 实现动态删除下拉列表框选项的代码实例。

【代码 7-14】（详见源代码目录 ch07-js-select-remove-option.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>动态删除下拉列表框选项</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center">
    <table>
        <tr>
            <th>设置颜色:  </th>
            <td>
                <!-- 定义下拉列表框 -->
                <select name="selSetColor" id="idSelSetColor">
                    <option>请选择...</option>
                    <option value="red">Red</option>
                    <option value="yellow">Yellow</option>
                    <option value="green">Green</option>
                </select>  
            </td>
        </tr>
        <tr>
            <td>删除选项:  </td>
            <input type="button" id="id-remove-color" value="删除颜色"
                    onclick="removeColor()"/>
        </tr>
        <tr>
```

```
32             <td>全部删除:  </td>
33             <input type="button" id="id-remove-all-color" value="全部删除"
                      onclick="removeAllColors()"/>
34          </tr>
35      </table>
36  </div>
37  </body>
38  <script type="text/javascript">
39      /**
40       * remove option color
41       * @param thisid
42       */
43      function removeColor(thisid) {
44          var idSel = document.getElementById('idSelSetColor');
45          idSel.options.remove(idSel.selectedIndex);
46      }
47      /**
48       * remove option colors
49       * @param thisid
50       */
51      function removeAllColors(thisid) {
52          var idSel = document.getElementById('idSelSetColor');
53          idSel.options.length = 0;
54      }
55  </script>
56  </html>
```

关于【代码 7-14】的说明：

- 第 19～24 行代码通过<select id="idSelSetColor">标签元素定义了一个下拉列表框，并初始化了一组颜色选项，用于实现“动态删除下拉列表框选项”的操作。
- 第 43～46 行代码定义的事件处理方法用于动态删除下拉列表框中的选项。其中，在第 45 行代码中，先通过 selectedIndex 属性获取当前选中的下拉列表框选项，再通过 remove()方法删除该选中项。
- 第 51～54 行代码定义的事件处理方法用于动态删除下拉列表框中的全部选项。其中，在第 53 行代码中，通过将 options 集合的 length 属性设置为 0 实现了删除全部选项的操作。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 7.14 所示。

如图 7.14 中箭头和标注所示，我们分别测试了删除颜色（Yellow）选项和删除全部颜色选项的操作。

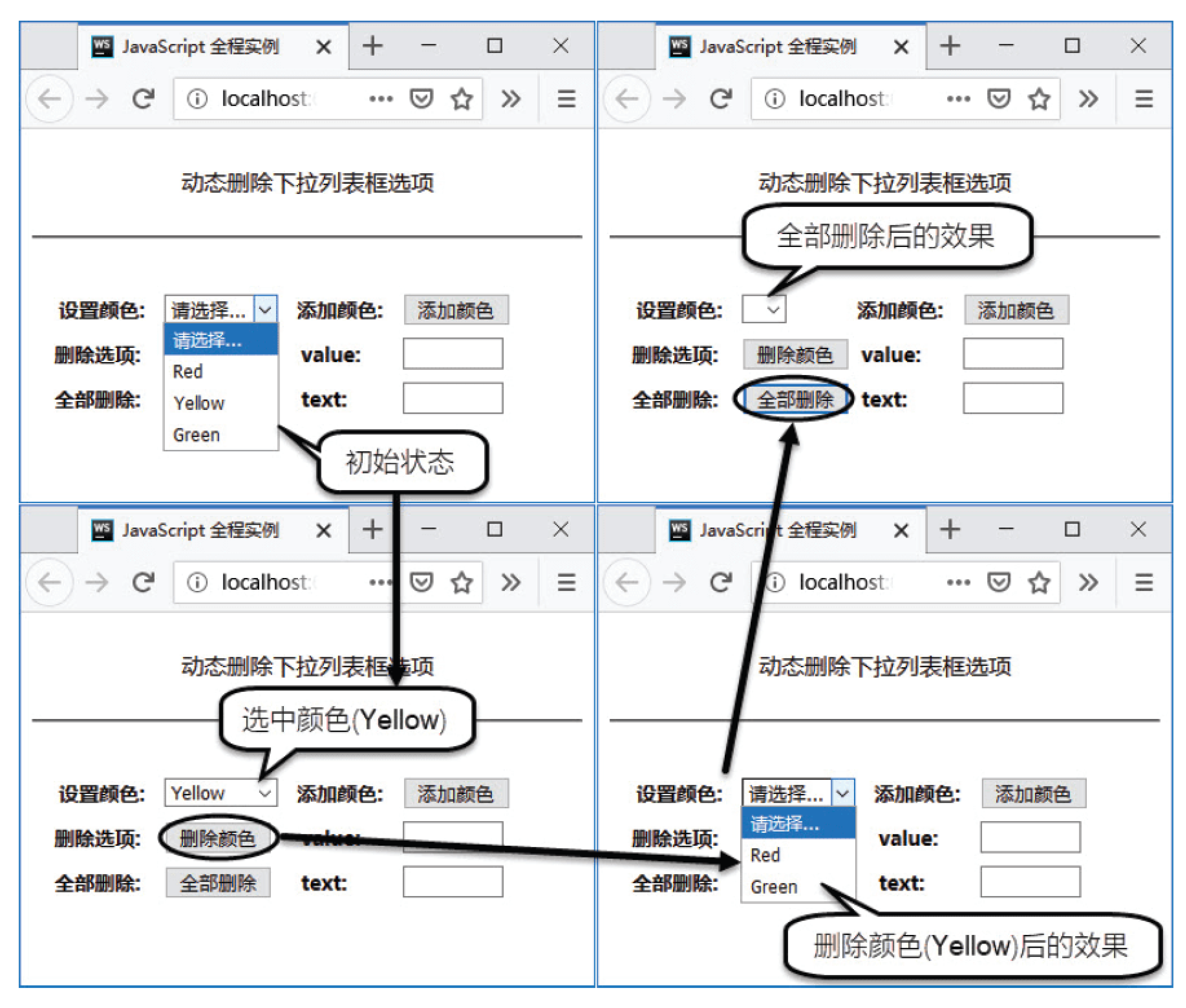


图 7.14　JavaScript 实现动态删除下拉列表框选项

7.14　二级联动下拉列表框

在 HTML 网页设计中，二级联动下拉列表框是一种比较常用的控件。所谓二级联动下拉列表框，就是指两个存在从属关系的下拉列表框。从属关系也可以称为上下级关系。在选择上一级下拉列表框的选项时，下一级下拉列表框的选项会随之进行切换。下面介绍一个通过 JavaScript 实现“大洲-国家”二级联动下拉列表框的代码实例。

【代码 7-15】（详见源代码目录 ch07-js-select-2-level.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>二级联动下拉列表框</nav>
```

```
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center">
    <form name="form2sel">
        大洲:
        <select name="continent" onchange="sel_country()">
            <option value="0">选择大洲...</option>
            <option value="亚洲">亚洲</option>
            <option value="欧洲">欧洲</option>
            <option value="美洲">美洲</option>
        </select>
        国家:
        <select name="country">
            <option value="0">选择国家...</option>
        </select>
    </form>
</div>
</body>
<script type="text/javascript">
    var country = [
        ["中国", "日本", "韩国"],
        ["英国", "德国", "法国"],
        ["美国", "巴西", "阿根廷"]
    ];
    function sel_country() {
        // TODO: 获得大洲下拉选择框对象
        var selContinent = document.form2sel.continent;
        // TODO: 获得国家下拉选择框对象
        var selCountry = document.form2sel.country;
        // TODO: 获得大洲所对应的国家数组
        var arrContinentCountry = country[selContinent.selectedIndex - 1];
        // TODO: 清空国家下拉列表框,仅保留第一个选项
        selCountry.length = 1;
        // TODO: 将国家数组中的值填充到国家下拉列表框中
        for (var i = 0; i < arrContinentCountry.length; i++) {
            selCountry[i + 1] = new Option(arrContinentCountry[i],
                                            arrContinentCountry[i]);
        }
    }
</script>
</html>
```

关于【代码 7-15】的说明：

- 第 16 ~ 21 行代码通过<select name="continent">标签元素定义了第一级“大洲”下拉列表框，初始化了一组“大洲”的选项，并声明了 onchange 事件处理方法（sel_country()），用于实现“二级联动下拉列表框”中的第一级下拉列表框。
- 第 23 ~ 25 行代码通过<select name="country">标签元素定义了第二级“国家”下拉列表框，仅仅初始化了一个“选择国家”选项，用于实现“二级联动下拉列表框”中的第二级下拉列表框。
- 第 30 ~ 34 行代码声明并初始化了一个二维数组（country），定义了“大洲”下所对应的“国家”数组。
- 第 35 ~ 48 行代码是事件处理方法（sel_country()）的实现过程。首先，第 41 行代码根据第一级“大洲”下拉列表框（<select name="continent">）的 selectedIndex 属性值，获取了“大洲”所对应的“国家”数组；然后，第 45 ~ 47 行通过 for 循环语句将获取的“国家”数组填充进第二级“国家”下拉列表框（<select name="country">），从而实现了“大洲-国家”二级联动下拉列表框。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 7.15 所示。

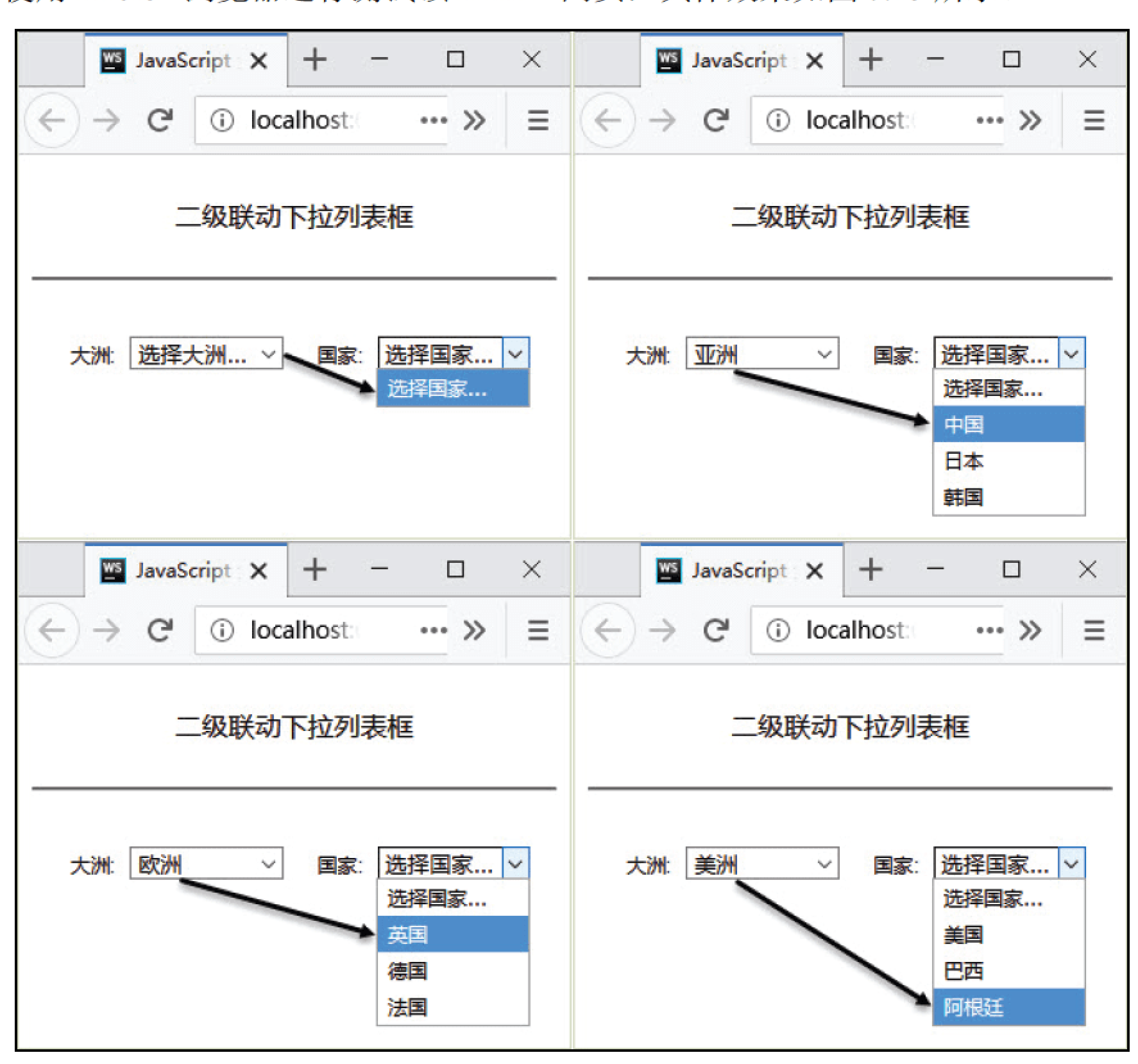


图 7.15　JavaScript 实现二级联动下拉列表框

7.15　三级联动下拉列表框

在本节中，我们将在二级联动下拉列表框的基础上进一步介绍如何实现三级联动下拉列表框。下面看一个通过 JavaScript 实现三级联动下拉列表框的代码实例。

【代码 7-16】（详见源代码目录 ch07-js-select-3-level.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>三级联动下拉列表框</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center">
    <form name="form3sel">
        大洲:
        <select name="continent" onchange="sel_country()">
            <option value="0">选择大洲...</option>
            <option value="亚洲">亚洲</option>
            <option value="欧洲">欧洲</option>
            <option value="美洲">美洲</option>
        </select>
        国家:
        <select name="country" onchange="sel_city()">
            <option value="0">选择国家...</option>
        </select>
        城市:
        <select name="city">
            <option value="0">选择城市...</option>
        </select>
    </form>
</div>
</body>
```

```
<script type="text/javascript">
    var country = [
        ["中国", "日本", "韩国"],
        ["英国", "德国", "法国"],
        ["美国", "巴西", "阿根廷"]
    ];
    var city = [
      [["北京", "上海", "广州"], ["东京", "大阪", "名古屋"], ["首尔", "仁川",
        "济州"]],
      [["伦敦", "曼彻斯特", "利物浦"], ["柏林", "慕尼黑", "法兰克福"], ["巴黎",
        "里尔", "摩纳哥"]],
      [["华盛顿", "纽约", "洛杉矶"], ["里约热内卢", "圣保罗", "巴西利亚"],
       ["布宜诺斯艾利斯", "萨尔多瓦", "罗萨里奥"]]
    ];
    function sel_country() {
        // TODO: 获得大洲下拉选择框对象
        var selContinent = document.form3sel.continent;
        // TODO: 获得国家下拉选择框对象
        var selCountry = document.form3sel.country;
        // TODO: 获得城市下拉选择框对象
        var selCity = document.form3sel.city;
        // TODO: 获得大洲所对应的国家数组
        var arrContinentCountry = country[selContinent.selectedIndex - 1];
        // TODO: 清空国家下拉列表框,仅保留第一个选项
        selCountry.length = 1;
        // TODO: 清空城市下拉列表框,仅保留第一个选项
        selCity.length = 1;
        // TODO: 将国家数组中的值填充到国家下拉列表框中
        for (var i = 0; i < arrContinentCountry.length; i++) {
              selCountry[i + 1] = new Option(arrContinentCountry[i],
             arrContinentCountry[i]);
        }
    }
    function sel_city() {
        // TODO: 获得大洲下拉选择框对象
        var selContinent = document.form3sel.continent;
        // TODO: 获得国家下拉选择框对象
        var selCountry = document.form3sel.country;
        // TODO: 获得城市下拉选择框对象
        var selCity = document.form3sel.city;
        // TODO: 获得大洲所对应的国家数组
```

```
70          var arrCountryCity = city[selContinent.selectedIndex-1]
                       [selCountry.selectedIndex-1];
71          // TODO: 清空国家下拉列表框,仅保留第一个选项
72          selCity.length = 1;
73          // TODO: 将国家数组中的值填充到国家下拉列表框中
74          for (var i = 0; i < arrCountryCity.length; i++) {
75              selCity[i + 1] = new Option(arrCountryCity[i],
                arrCountryCity[i]);
76          }
77      }
78  </script>
79  </html>
```

关于【代码 7-16】的说明：

- 第 16～21 行代码通过<select name="continent">标签元素定义了第一级“大洲”下拉列表框，初始化了一组“大洲”的选项，并声明了 onchange 事件处理方法（sel_country()），用于实现“三级联动下拉列表框”中的第一级下拉列表框。
- 第 23～25 行代码通过<select name="country">标签元素定义了第二级“国家”下拉列表框，仅仅初始化了一个“选择国家”选项，并声明了 onchange 事件处理方法（sel_city()），用于实现“三级联动下拉列表框”中的第二级下拉列表框。
- 第 27～29 行代码通过<select name="city">标签元素定义了第三级“城市”下拉列表框，仅仅初始化了一个“选择城市”选项，用于实现“三级联动下拉列表框”中的第三级下拉列表框。
- 第 34～38 行代码声明并初始化了一个二维数组（country），定义“大洲”下所对应的“国家”数组。
- 第 39～43 行代码声明并初始化了一个三维数组（city），定义“国家”下所对应的“城市”数组。
- 第 44～61 行代码是事件处理方法（sel_country()）的实现过程。首先，第 52 行代码根据第一级“大洲”下拉列表框（<select name="continent">）的 selectedIndex 属性值获取了“大洲”所对应“国家”的数组；然后，第 58～60 行通过 for 循环语句将获取的“国家”数组填充进第二级“国家”下拉列表框（<select name="country">），从而实现了三级联动下拉列表框中的“大洲-国家”二级联动功能。
- 第 62～77 行代码是事件处理方法（sel_city()）的实现过程。首先，第 70 行代码根据第一级“大洲”下拉列表框（<select name="continent">）的 selectedIndex 属性值以及第二级“国家”下拉列表框（<select name="country">）的 selectedIndex 属性值，获取了“大洲”所对应“国家”以及“国家”所对应“城市”的数组；然后，第 74～76 行通过 for 循环语句将获取的“城市”数组填充进第三级“城市”下拉列表框（<select name="city">），从而实现了三级联动下拉列表框中的“国家-城市”二级联动功能。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果分别如图 7.16、图 7.17 和图 7.18 所示。

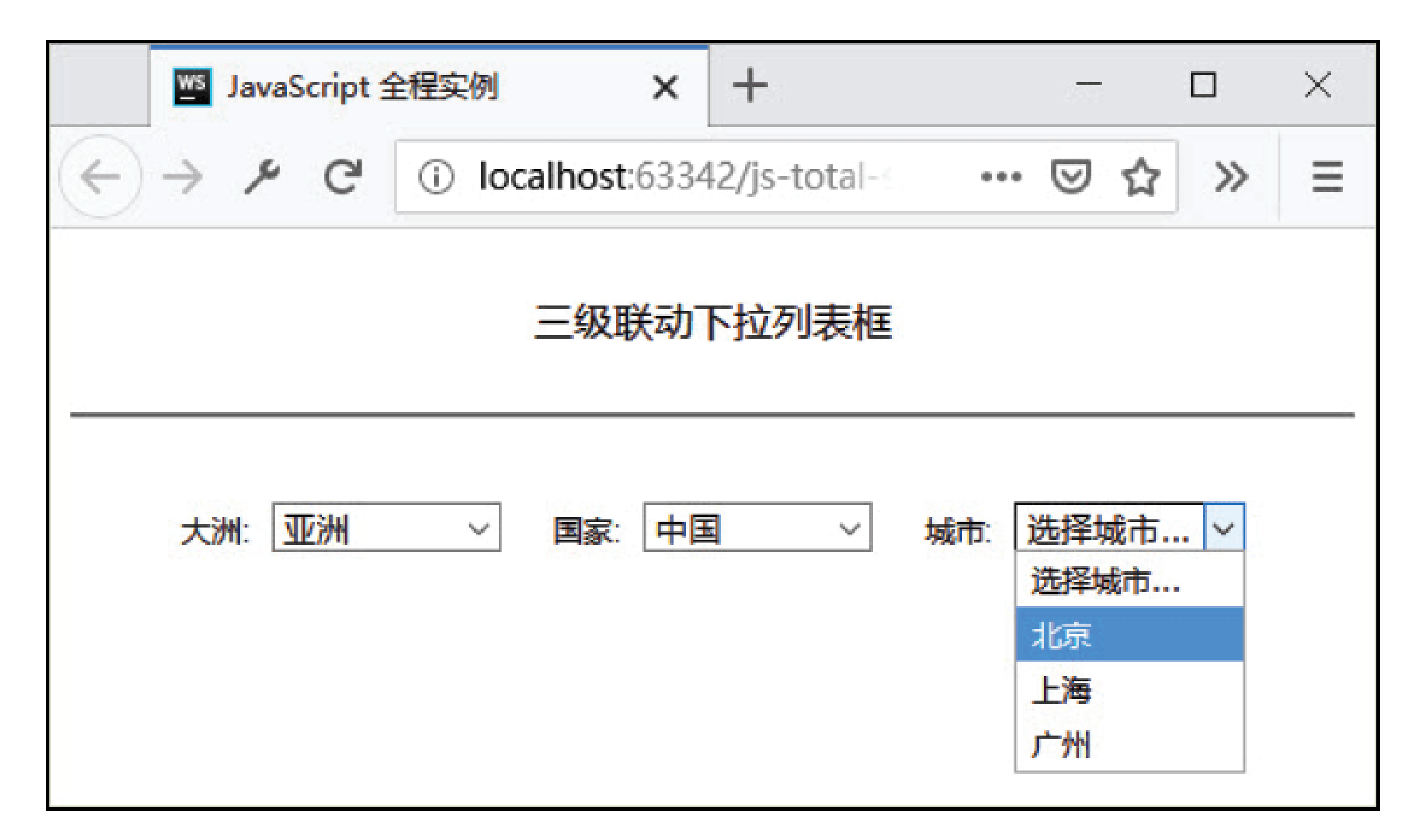


图 7.16　JavaScript 实现三级联动下拉列表框（一）

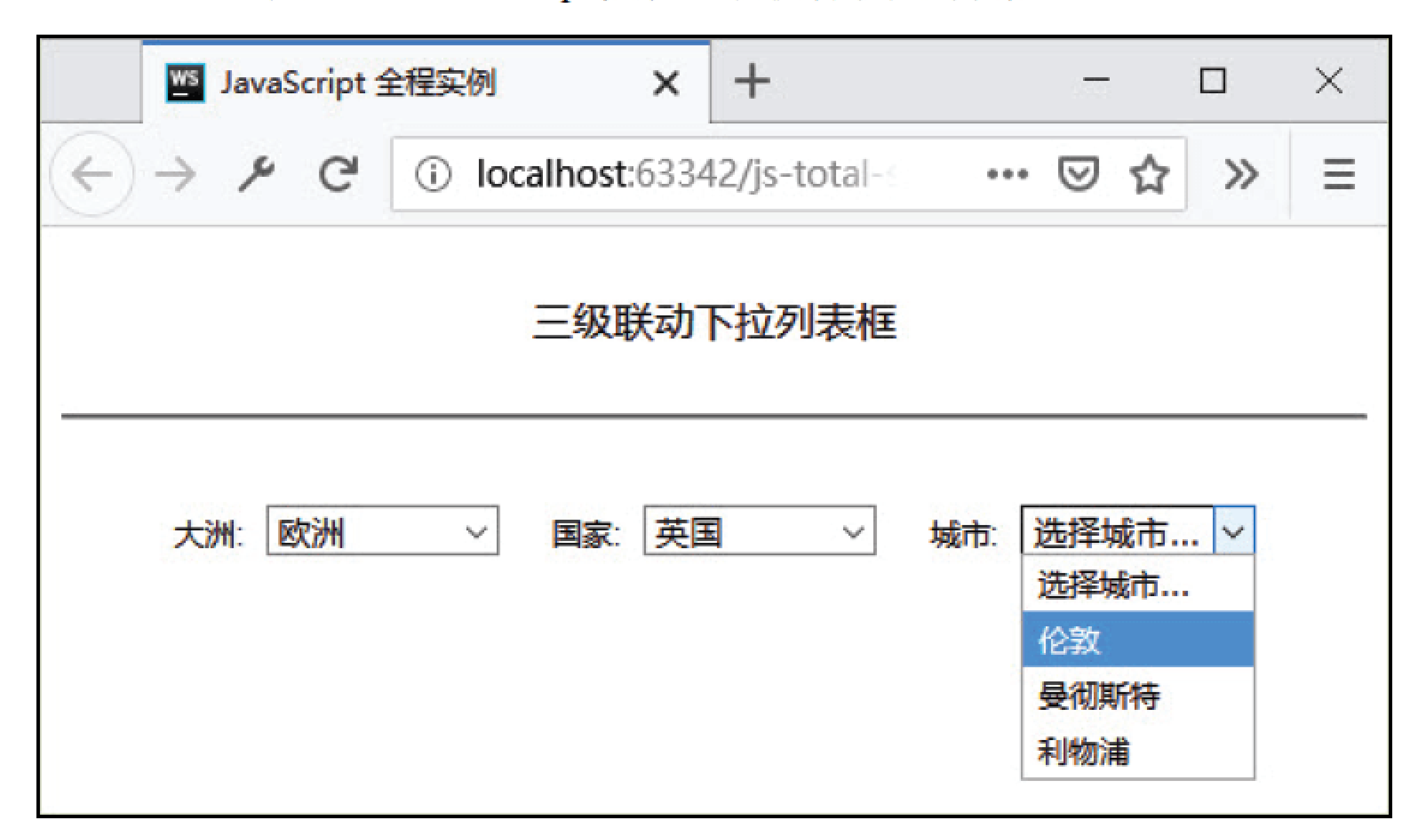


图 7.17　JavaScript 实现三级联动下拉列表框（二）

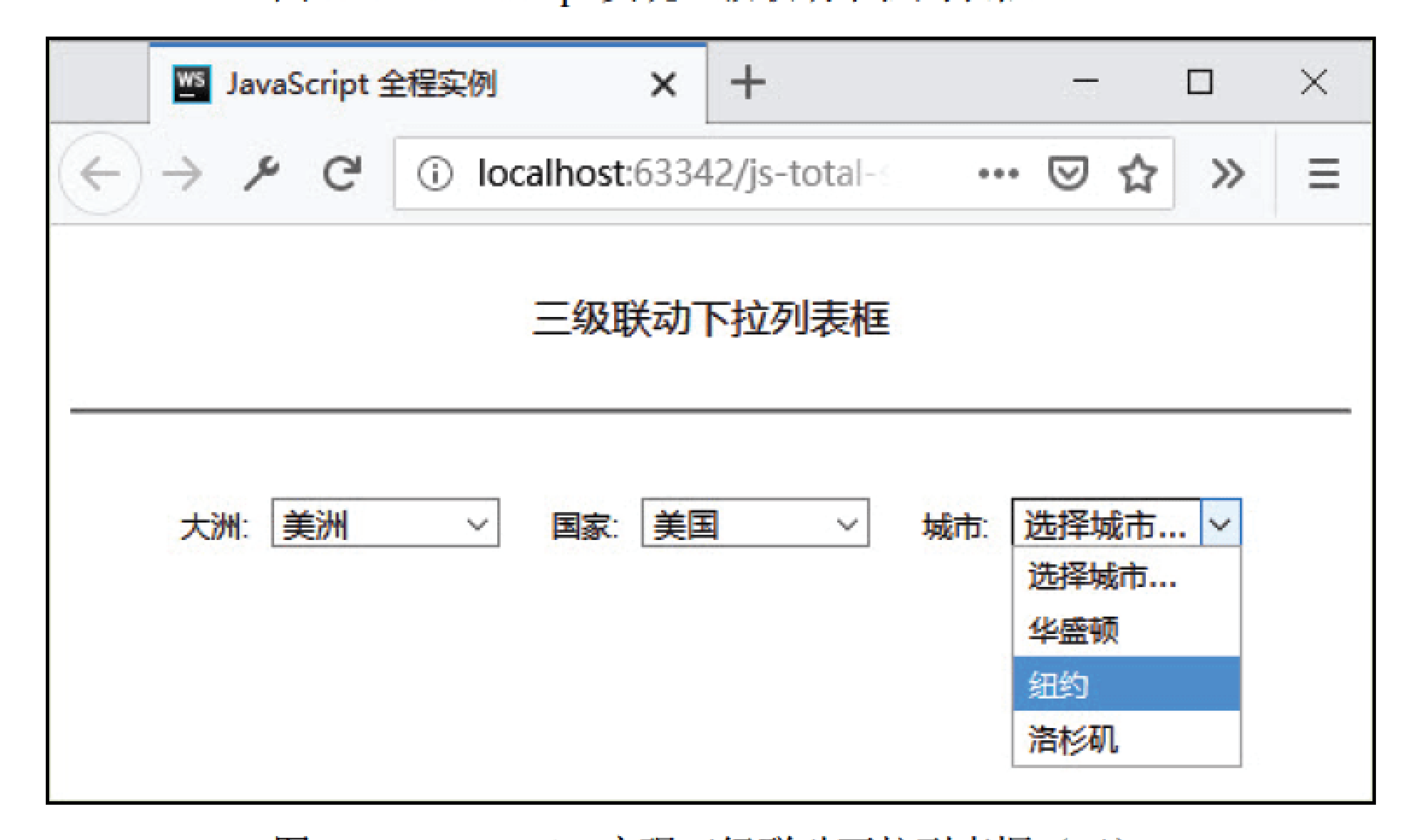


图 7.18　JavaScript 实现三级联动下拉列表框（三）

7.16　可输入的下拉列表框

在 HTML DOM 规范标准中，下拉列表框<select>控件是仅可以选择，但是不可以输入的。于是，如何通过扩展标准下拉列表框<select>的功能，让其既可以支持用户下拉选择，也可以支持用户人工输入，就成为设计人员展现智慧的舞台了。

虽然目前业内已经开发出了多款 JavaScript 框架或插件来实现这种可输入的下拉列表框了，但是这里还是想介绍一种通过最基础的 JavaScript + CSS 代码方式来实现的可输入的下拉列表框。这种方式的设计思路很简单，先通过将一个文本框覆盖在一个下拉列表框上，再将用户选择的下拉列表框选项填充到文本框中即可。

下面介绍一个通过 JavaScript 实现可输入的下拉列表框的代码实例。

【代码 7-17】（详见源代码目录 ch07-js-input-select.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<style type="text/css">
    div#id-div-center {
        position: relative;
        width: 320px;
        height: auto;
        border: 0px solid black;
        margin: 32px auto;
        padding: 2px;
    }
    div#id-div-center span {
        overflow: hidden;
        margin-left: 100px;
        width: 18px;
    }
    div#id-div-center select {
        width: 128px;
        margin-left: -100px;
    }
    div#id-div-center input {
        position: absolute;
        width: 105px;
```

```
28          height: 14px;
29          left: 98px;
30      }
31  </style>
32  <body>
33  <!-- 添加文档主体内容 -->
34  <header>
35      <nav>可输入的下拉列表框</nav>
36  </header>
37  <!-- 添加文档主体内容 -->
38  <div id="id-div-center" style="">
39      <span style="">
40          <select id="id-sel" onchange="sel_input(this.id);">
41              <option value="html">HTML</option>
42              <option value="js">JavaScript</option>
43              <option value="css">CSS</option>
44          </select>
45      </span>
46      <input id="id-input" style="">
47  </div>
48  </body>
49  <script type="text/javascript">
50      function sel_input(thisid) {
51          var sel = document.getElementById(thisid);
52          var input = document.getElementById("id-input");
53          input.value = sel.options[sel.selectedIndex].text;
54      }
55  </script>
56  </html>
```

关于【代码 7-17】的说明：

- 第 38～47 行代码通过<div>标签元素定义了一个顶层容器（CSS 样式定义见第 08～15 行代码），用于包含下拉列表框和文本框。
    - 第 39～45 行代码通过<span>标签元素定义了一个二级容器（CSS 样式定义见第 16～20 行代码），用于包含下拉列表框。其中，第 40～44 行代码通过<select id="id-sel">标签元素定义了“可输入的下拉列表框”中的下拉列表框部分，并添加了 onchange 事件处理方法（sel_input()）。
    - 第 46 行代码通过<input>标签元素定义了一个文本框，用于实现“可输入的下拉列表框”中的可输入部分。

- 第 50 ~ 54 行代码是事件处理方法（sel_input()）的实现过程。首先，在第 53 行代码中，先获取用户选择的下拉列表框选项，再将选项文本填充进文本框，实现了“可输入的下拉列表框”中的用户选择功能。其次，第 46 行代码定义的文本框自身就具有输入功能。这两项功能结合在一起，就实现了“可输入的下拉列表框”功能。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果分别如图 7.19 和图 7.20 所示。

如图 7.20 中箭头所示，这里演示了可输入的下拉列表框中的用户输入（人工输入的文本为“可输入”）功能。

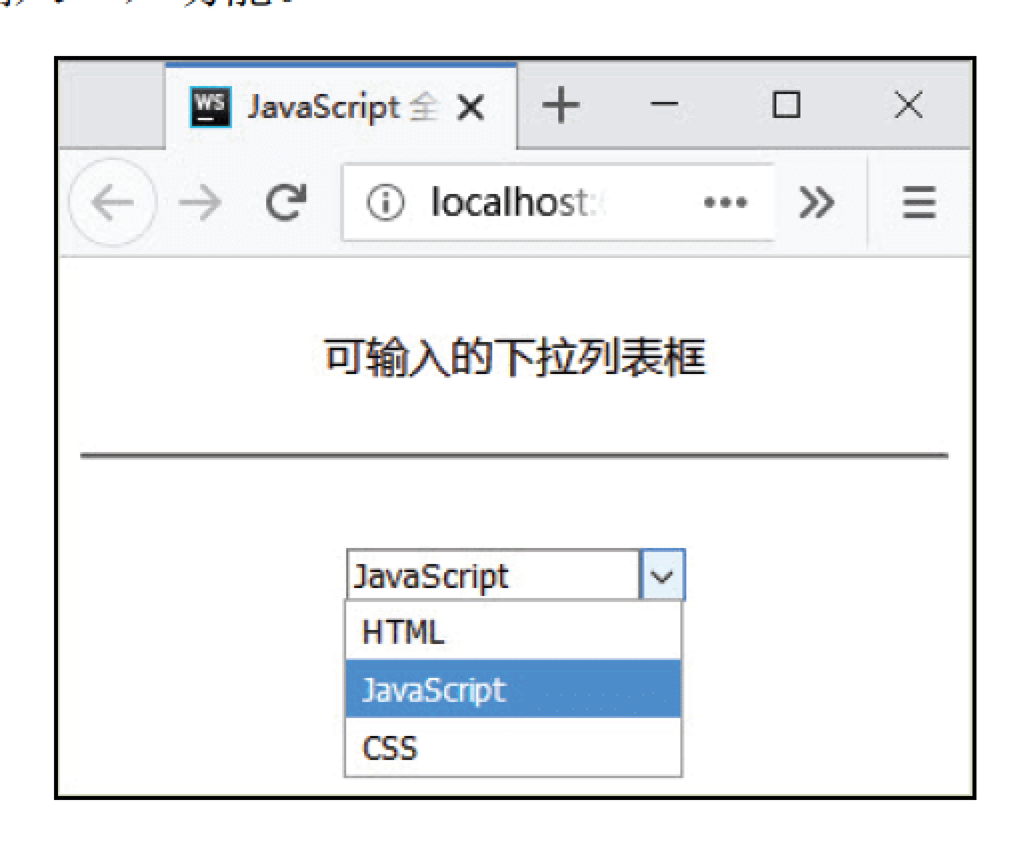


图 7.19　JavaScript 实现可输入的下拉列表框（选择功能）

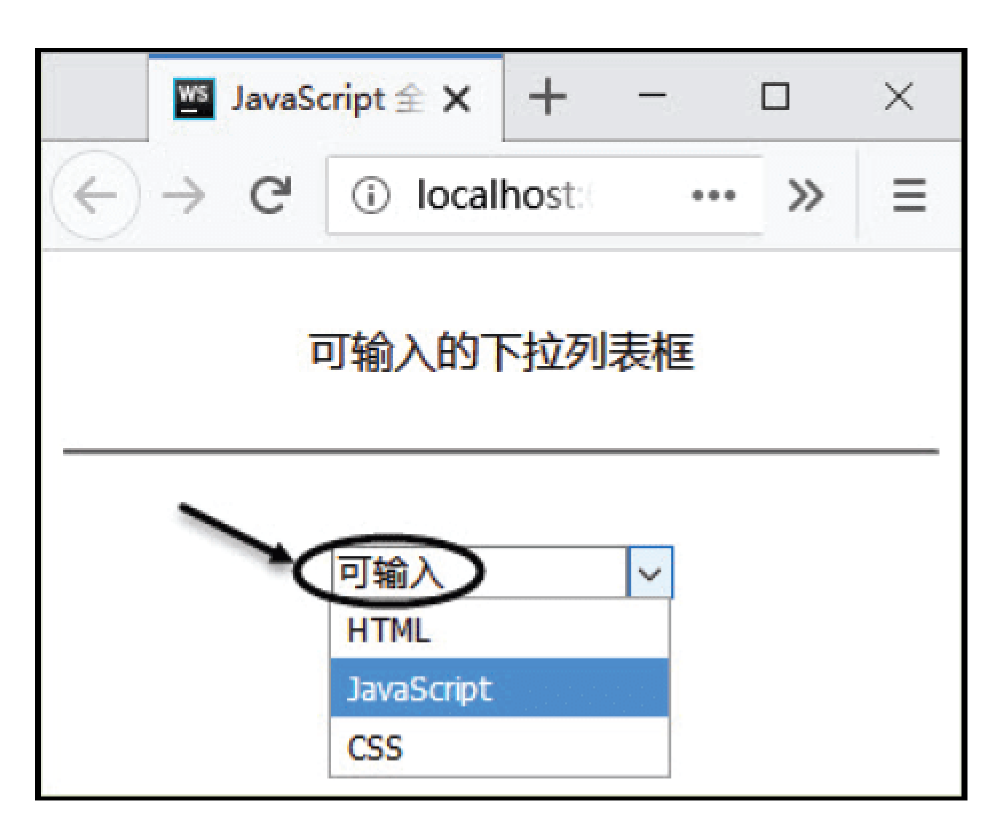


图 7.20　JavaScript 实现可输入的下拉列表框（输入功能）

第 8 章　日期和时间特效

本章介绍如何通过 JavaScript 来实现日期和时间的各种特效，通过这些特效来丰富 HTML 页面的设计手段和展示效果。

8.1　日期和时间概述

日期和时间是 HTML 页面中不太显眼，但是会经常使用到的元素。其实，日期和时间在 JavaScript 脚本语言开发中是属于同一个概念的。在 JavaScript 对象中，专门为日期和时间定义了一个 Date 对象，该对象用来处理日期和时间的编程。

关于 Date 对象的语法格式如下：

【代码 8-1】

```
/** Date 对象定义 */
var myDate=new Date();
```

另外，默认情况下 Date 对象会自动把当前日期和时间定义为初始化值。

通过 JavaScript Date 对象可以为日期和时间实现多种页面特效，比如标题栏日期、根据时间段动态显示标题、倒计时效果、时钟特效、日历特效、定时器应用，等等。本章将为读者介绍多种日期和时间特效的代码实例。

8.2　在标题栏显示当前日期

HTML 页面的标题栏一般用来显示文档标题，不过也有设计人员会将日期和时间显示在标题栏中。其实，在标题栏显示当前日期的操作非常简单，只需要将日期和时间重新设定给 Document 对象的标题“title”属性就可以了。下面介绍一个通过 JavaScript 实现在标题栏显示当前日期的代码实例。

【代码 8-2】（详见源代码目录 ch08-js-date-title.html 文件）

```
<!doctype html>
<html lang="en">
<head>
```

```
04      <!-- 添加文档头部内容 -->
05      <title>JavaScript 全程实例</title>
06  </head>
07  <body>
08  <!-- 添加文档主体内容 -->
09  <header>
10      <nav>在标题栏显示当前日期</nav>
11  </header>
12  <!-- 添加文档主体内容 -->
13  </body>
14  <script type="text/javascript">
15      window.onload = function () {
16          var date = new Date();
17          document.title = date.toString();
18      }
19  </script>
20  </html>
```

关于【代码 8-2】的说明：

- 第 16 行代码通过 Date 对象定义了一个日期和时间变量（date）。
- 第 17 行代码通过 Document 对象的 title 属性，将当前时间（date）添加在标题栏（title 对象）中。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 8.1 所示。

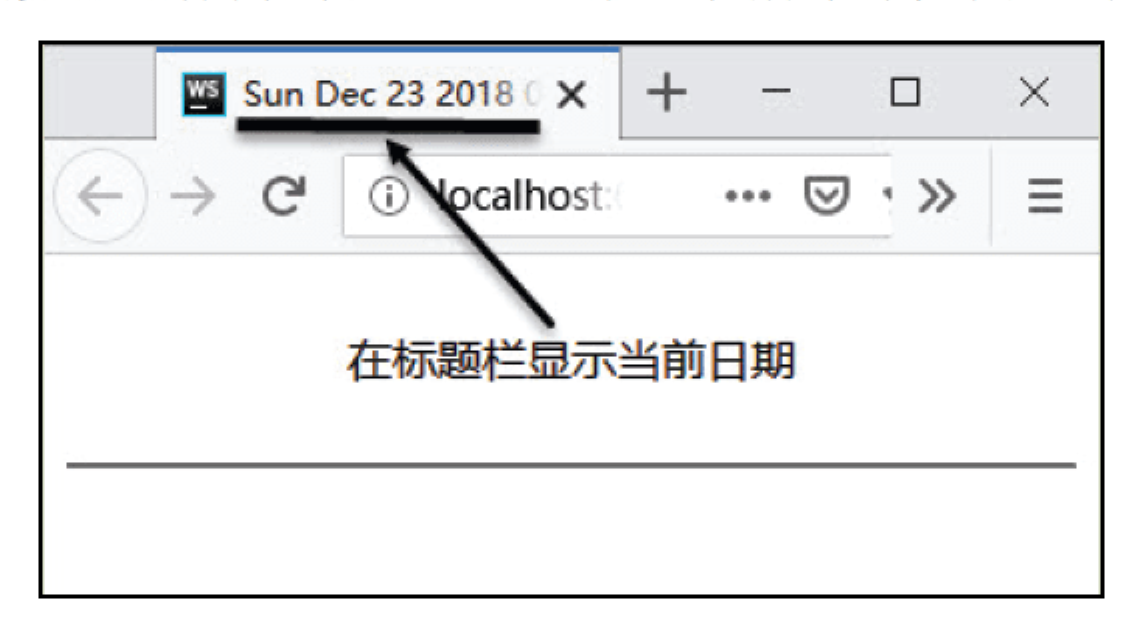


图 8.1　JavaScript 实现在标题栏显示当前日期

如图 8.1 箭头所示，页面标题栏中成功显示了通过 JavaScript 脚本语言获取的当前日期。

8.3　根据时间动态显示标题欢迎词

在 8.2 节中，我们介绍了"在标题栏显示当前日期"的 JavaScript 语言代码实现。本节将在前一节的基础上继续介绍"根据时间动态显示标题欢迎词"的方法。下面看一个具体的 JavaScript 代码实例。

【代码 8-3】（详见源代码目录 ch08-js-date-dyn-title.html 文件）

```
01 <!doctype html>
02 <html lang="en">
03 <head>
04     <!-- 添加文档头部内容 -->
05     <title>JavaScript 全程实例</title>
06 </head>
07 <body>
08 <!-- 添加文档主体内容 -->
09 <header>
10     <nav>根据时间动态显示标题欢迎词</nav>
11 </header>
12 <!-- 添加文档主体内容 -->
13 </body>
14 <script type="text/javascript">
15     window.onload = function () {
16         var date = new Date();
17         var hour = date.getHours();
18         if ((hour >= 0) && (hour < 6)) {
19             document.title = "凌晨好!";
20         } else if ((hour >= 6) && (hour < 8)) {
21             document.title = "早晨好!";
22         } else if ((hour >= 8) && (hour < 12)) {
23             document.title = "上午好!";
24         } else if ((hour >= 12) && (hour < 14)) {
25             document.title = "中午好!";
26         } else if ((hour >= 14) && (hour < 18)) {
27             document.title = "下午好!";
28         } else if ((hour >= 18) && (hour < 24)) {
29             document.title = "晚上好!";
30         } else {
31             document.title = date.toString();
32         }
33     }
34 </script>
35 </html>
```

关于【代码 8-3】的说明：

- 在第 17 行代码中先通过使用 Date 对象的 getHours()方法获取了当前时间的小时字段（hour）。
- 在第 18～32 行代码中通过 if 条件选择语句判断小时字段（hour）落在什么时间段，再将所对应的“标题欢迎词”通过 Document 对象的 title 属性添加在标题栏（title）中。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 8.2 所示。

图 8.2　JavaScript 实现根据时间动态显示标题欢迎词

如图 8.2 中箭头和标识所示，三个页面中分别演示了根据三个不同时间段动态显示标题欢迎词（“上午好” “中午好”和“下午好”）的效果。

8.4　根据月份动态显示背景

在 8.3 节中，我们介绍了“根据时间动态显示标题欢迎词”的 JavaScript 语言代码实现。本节将在前一节的基础上，继续介绍一种“根据月份动态显示背景”的方法。下面看一个具体的 JavaScript 代码实例。

【代码 8-4】（详见源代码目录 ch08-js-date-month-bg.html 文件）

```
<!doctype html>
<html lang="en">
```

```
03 <head>
04     <!-- 添加文档头部内容 -->
05     <title>JavaScript 全程实例</title>
06 </head>
07 <body>
08 <!-- 添加文档主体内容 -->
09 <header>
10     <nav>根据月份改变背景</nav>
11 </header>
12 <!-- 添加文档主体内容 -->
13 <div id="id-div-center" style="">
14     <span>
15         <img id="id-img-bg" src="" alt="正在加载背景...">
16     </span>
17 </div>
18 </body>
19 <script type="text/javascript">
20     window.onload = function () {
21         var bgImage = document.getElementById("id-img-bg");
22         var date = new Date();
23         var month = date.getMonth() + 1;
24         if ((month >= 1) && (month <= 12)) {
25             bgImage.src = "images/calendar-" + month + ".jpg";
26         }
27     }
28 </script>
29 </html>
```

关于【代码 8-4】的说明：

- 第 13 ~ 17 行代码通过<div>标签元素定义了一个层，其中第 15 行代码通过<img id="id-img-bg">标签元素定义了一幅背景图片，目标是根据月份动态显示不同的背景图片。
- 第 23 行代码通过使用 Date 对象的 getHours()方法获取了当前时间的月份字段。注意，getHours()方法的返回值范围为数值 0 ~ 11，因此返回值要再加 1 才会得到正确的月份（month）。
- 第 25 行代码通过将月份（month）数值与背景图片文件名连接在一起，实现“根据月份动态显示背景”的效果。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 8.3 所示。

如图 8.3 中箭头所示，两个页面中分别演示根据月份动态显示背景（“2018 年 12 月”和“2019 年 1 月”）的效果。

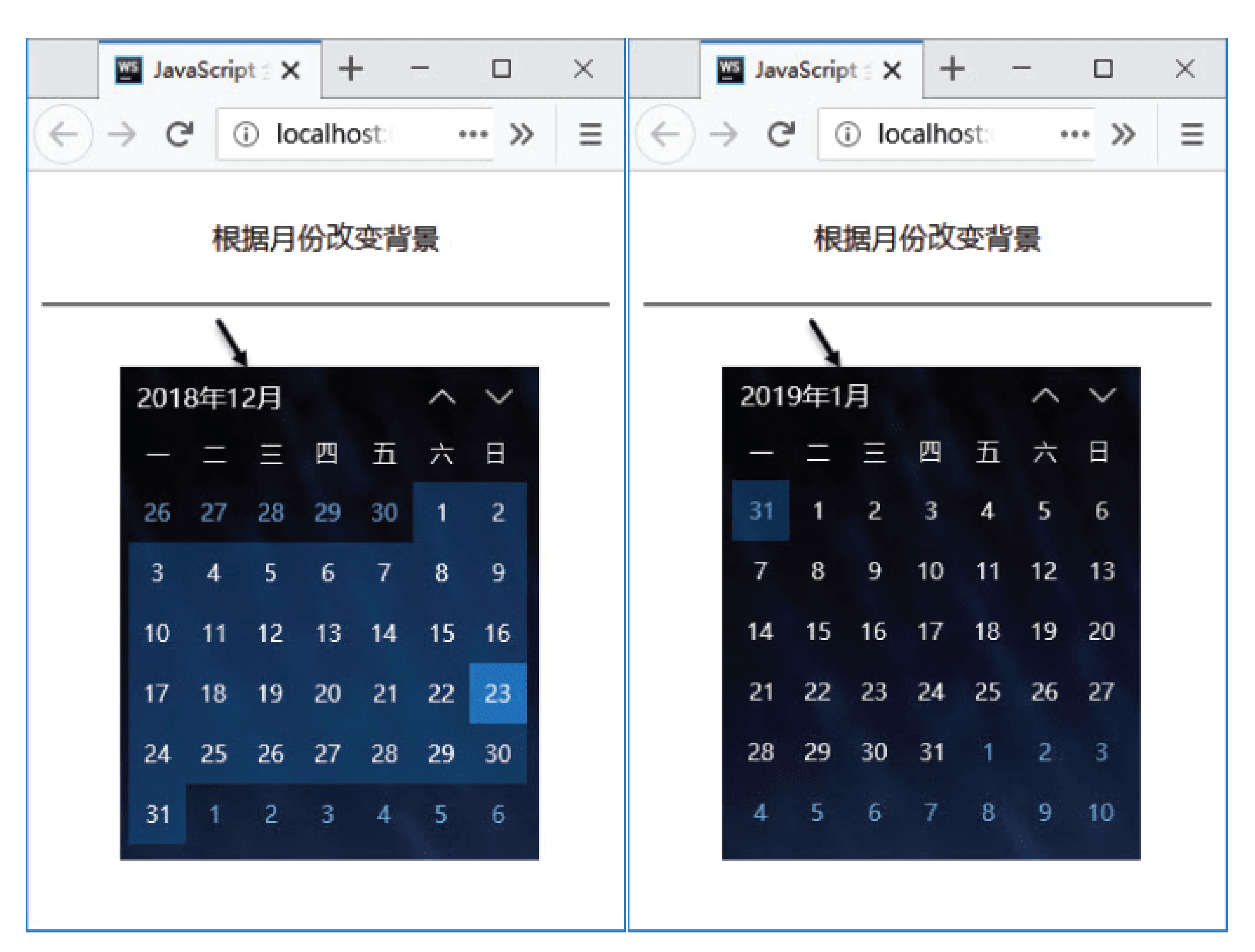


图 8.3　JavaScript 实现根据月份动态显示背景

8.5　格式化日期的方法

在 JavaScript 脚本语言中，直接通过 Date 对象获取的日期和时间是格林威治标准时间格式，虽然这个格式看上去很酷很专业，但是对于普通用户的可读性并不是很好。其实，通过 Date 对象提供的一系列方法再配合人工编程方式，完全可以将获取的日期和时间对象格式化成我们习惯的、可读性很好的格式。下面介绍一种简单、常用的日期格式化（YYYY-MM-DD hh:mm:ss）操作的 JavaScript 代码实例。

【代码 8-5】（详见源代码目录 ch08-js-date-format.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 日期和时间特效 - 格式化日期的方法</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
    Today:
    <span id="id-span-today"></span>
```

```
    格式化时间:
    <span id="id-span-format-date"></span>
</div>
</body>
<script type="text/javascript">
    window.onload = function () {
        var date = new Date();
        var today = document.getElementById("id-span-today");
        var formatToday = document.getElementById("id-span-format-date");
        today.innerText = date.toString();
        formatToday.innerText = formatDate(date);
    };
    function formatDate(d) {
        var curYear = d.getFullYear();
        var curMonth = d.getMonth() + 1;
        if (curMonth >= 1 && curMonth <= 9) {
            curMonth = "0" + curMonth;
        }
        var curDate = d.getDate();
        if (curDate >= 0 && curDate <= 9) {
            curDate = "0" + curDate;
        }
        var curHour = d.getHours();
        if (curHour >= 0 && curHour <= 9) {
            curHour = "0" + curHour;
        }
        var curMinutes = d.getMinutes();
        if (curMinutes >= 0 && curMinutes <= 9) {
            curMinutes = "0" + curMinutes;
        }
        var curSecond = d.getSeconds();
        if (curSecond >= 0 && curSecond <= 9) {
            curSecond = "0" + curSecond;
        }
        var arrWeek = ["星期日", "星期一", "星期二", "星期三", "星期四",
                  "星期五", "星期六"];
        var curDay = arrWeek[d.getDay()];
        // TODO: return format date
        return curYear + "-" + curMonth + "-" + curDate + " " + curHour + ":"
                  + curMinutes + ":" + curSecond + " " + curDay + ".";
    }
</script>
</html>
```

关于【代码 8-5】的说明：

- 第 15 行和第 17 行代码分别通过<span>标签元素定义了两个行内容器，分别用于显示正常未格式化的日期和格式化后的日期。
- 第 28 ~ 54 行代码定义的 formatDate()方法用于实现对正常日期的格式化操作。其中，在第 29 行代码中通过 getFullYear()方法获取了年份字段，在第 30 行代码中通过 getMonth()方法获取了月份字段（注意加 1 操作），在第 34 行代码中通过 getDate()方法获取了日期字段，在第 38 行代码中通过 getHours()方法获取了小时字段，在第 42 行代码中通过 getMinutes()方法获取了分钟字段，在第 46 行代码中通过 getSeconds()方法获取了秒数字段。另外，对于一位数字的日期和时间字段，均统一转换为了两位数字，这样的显示效果会更美观。第 50 行代码定义了一个从星期日到星期六的数组（arrWeek），并在第 51 行代码中通过 getDay()方法获取当前日期是星期几。第 53 行代码通过将前面获取的一系列关于日期和时间的信息，按照格式化日期（YYYY-MM-DD hh:mm:ss）的方式连接到一起，并作为 formatDate()方法的返回值。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 8.4 所示。

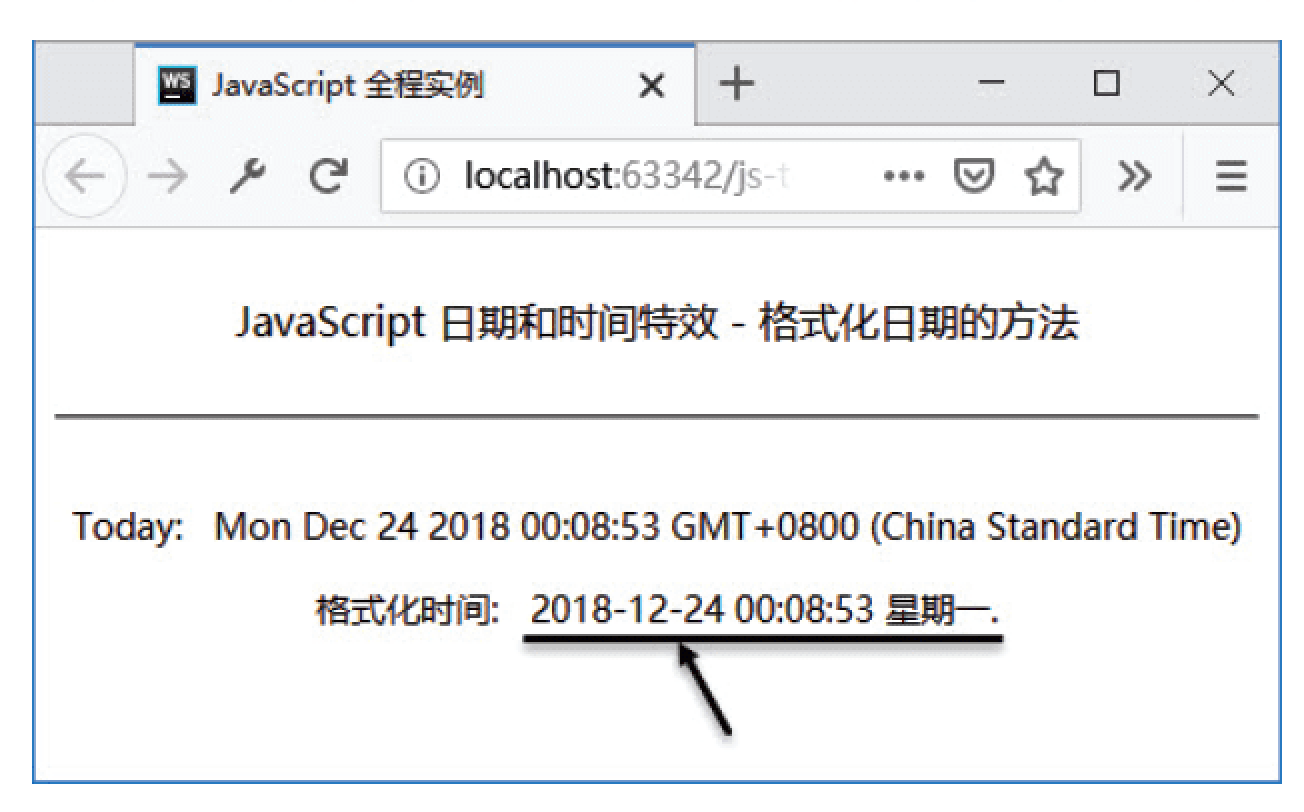


图 8.4　JavaScript 实现格式化日期的方法

如图 8.4 中箭头所示，页面中分别演示了标准格林尼治时间以及格式化（YYYY-MM-DD hh:mm:ss）日期和时间的效果。

8.6　判断今天是否为节假日

在很多日历应用（App）中，如果今天是节假日（包括双休日和国家法定节假日）在显示效果上均会有所区别。其实，通过 JavaScript 脚本语言可以有效地判断出双休日，然后针对国家法定节假日设定特殊的算法。下面介绍一个通过 JavaScript 实现判断今天是否为节假日的代码实例。

【代码 8-6】（详见源代码目录 ch08-js-date-holiday.html 文件）

```
<!doctype html>
<html lang="en">
<head>
```

```
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>判断今天是否为节假日</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
    今天是:<span id="id-span-date"></span>
    今天是:<span id="id-span-hw"></span>
</div>
</body>
<script type="text/javascript">
    window.onload = function () {
        var today = document.getElementById("id-span-date");
        var holiday_weekend = document.getElementById("id-span-hw");
        var date = new Date();
        today.innerText = curDate(date);
        holiday_weekend.innerText = getHolidayWeekend(date);
    };
    function curDate(d) {
        var arrWeek = ["星期日", "星期一", "星期二", "星期三", "星期四",
                      "星期五", "星期六"];
        var curYear = d.getFullYear();
        var curMonth = d.getMonth() + 1;
        var curDate = d.getDate();
        var curDay = arrWeek[d.getDay()];
        return curYear + "-" + curMonth + "-" + curDate + " " + curDay + ".";
    }
    function getHolidayWeekend(d) {
        var vHolidayWeekend;
        var hMonth = d.getMonth() + 1;
        var hDate = d.getDate();
        var hDay = d.getDay();
        if (hDate == 1) {
            if (hMonth == 1) {
                vHolidayWeekend = "新年元旦";
            } else if (hMonth == 5) {
                vHolidayWeekend = "五一劳动节";
            } else if (hMonth == 10) {
                vHolidayWeekend = "十一国庆节";
```

```
46              } else {
47              }
48          } else if ((hMonth == 3) && (hDate == 8)) {
49              vHolidayWeekend = "三八妇女节";
50          } else if ((hMonth == 4) && (hDate == 5)) {
51              vHolidayWeekend = "清明节";
52          } else if ((hMonth == 8) && (hDate == 15)) {
53              vHolidayWeekend = "中秋节";
54          } else {
55              if(isWeekend(d)) {
56                  vHolidayWeekend = "周末(六日)";
57              } else {
58                  vHolidayWeekend = "工作日";
59              }
60          }
61          return vHolidayWeekend;
62      }
63      function isWeekend(d) {
64          var hDay = d.getDay();
65          if ((hDay == 0) || (hDay == 6)) {
66              return true;
67          } else {
68              return false;
69          }
70      }
71  </script>
72  </html>
```

关于【代码 8-6】的说明：

- 第 14 ~ 15 行代码通过<span>标签元素定义了两个行内容器，第一个用于显示当前日期，第二个用于显示当前日期是否为节假日的判断结果。
- 第 26 ~ 33 行代码定义的 curDate()方法用于获取当前日期（格式为 YYYY-MM-DD Week）。
- 第 34 ~ 62 行代码定义的 getHolidayWeekend()方法用于判断当前日期是否为节假日，这里的节假日包括元旦、五一劳动节、十一国庆节、三八妇女节、清明节和中秋节，还包括正常的周末（周六和周日）休息日。因为类似元旦或五一劳动节这类的节假日都是有固定日期的，所以在第 19 ~ 54 行代码中通过判断当前日期是否为这些固定日期来甄别是否为节假日。而在第 55 ~ 59 行代码中，通过调用 isWeekend()方法来判断当前日期是否为周末休息日。
- 第 63 ~ 70 行代码定义的是 isWeekend()方法的实现过程。其中，第 64 行代码通过调用 getDay()方法获取了当前日期是星期几，然后第 65 ~ 69 行代码判断是否为周六或周日（通过布尔值返回）。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 8.5 所示。

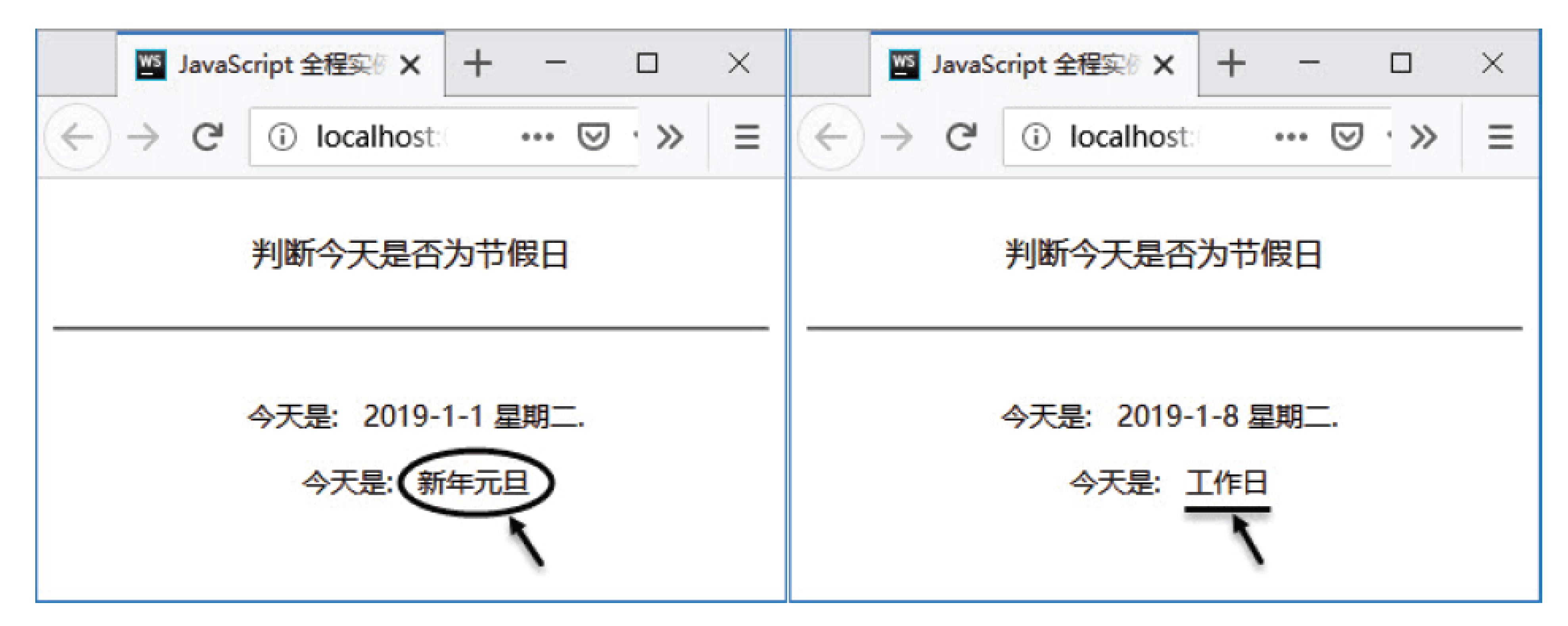


图 8.5　JavaScript 实现判断今天是否为节假日

如图 8.5 中箭头所示，两个页面中分别演示了新年元旦（2019 年 1 月 1 日）和平时工作日（2019 年 1 月 8 日）的效果。

8.7　每秒刷新的时间展示效果

通过 JavaScript 的 Date 对象可以获取详细的时间参数，如果结合使用 setInterval()函数进行定时，就可以在页面中模拟出每秒刷新的时间展示效果。下面看一个具体的 JavaScript 代码实例。

【代码 8-7】（详见源代码目录 ch08-js-date-timer.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>每秒刷新的时间展示效果</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
    每秒刷新的时间展示效果:
    <span id="id-span-timer"></span>
</div>
</body>
<script type="text/javascript">
    window.onload = function () {
        // TODO: 使用 setInterval 函数进行定时
```

```
21         window.setInterval(function () {
22             // TODO: 得到当前的时间对象
23             var date = new Date();
24             var str = '';        // TODO: 定义拼接的字符变量
25             // TODO: 得到小时数
26             str += date.getHours();
27             str += ':';          // TODO: 拼接冒号
28             // TODO: 得到分钟数
29             str += date.getMinutes();
30             str += ':';          // TODO: 拼接冒号
31             // TODO: 得到秒数
32             str += date.getSeconds();
33             // TODO: 把结果显示出来
34             document.getElementById('id-span-timer').innerHTML = str;
35         }, 1000);   // TODO: 间隔为 1 秒
36     };
37 </script>
38 </html>
```

关于【代码 8-7】的说明：

- 第 15 行代码通过<span>标签元素定义了一个行内容器，用于实现每秒刷新的时间展示效果。
- 第 21 ~ 35 行代码通过调用 Window 对象的 setInterval()方法进行定时，时间间隔定义为 1 秒。其中，第 26 行、第 29 行和第 32 行代码分别通过调用 getHours()方法、getMinutes()方法和 getSeconds()方法获取了小时、分钟和秒数，并通过 str 变量连接为一个时间格式（hh:mm:ss）。借助于 setInterval()方法设定 1 秒定时，因此在页面中实现每秒刷新的时间展示效果。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 8.6 所示。

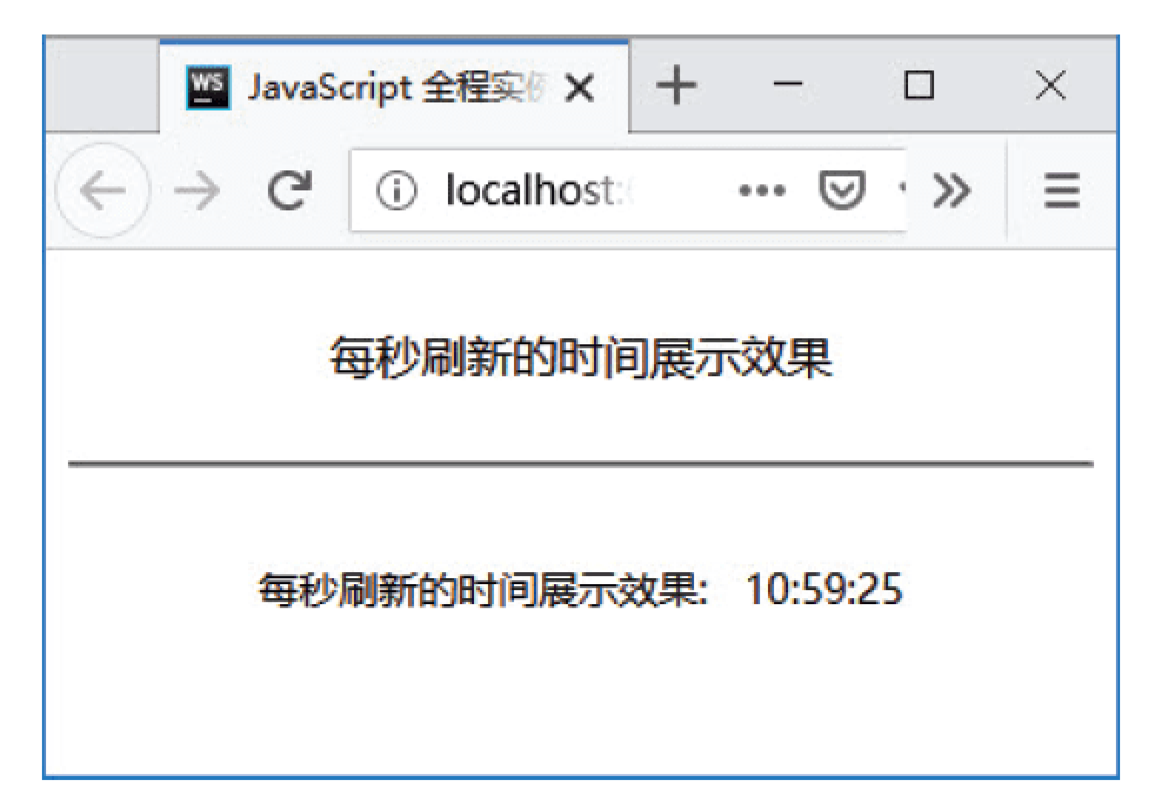


图 8.6　JavaScript 实现每秒刷新的时间展示效果

8.8 时间计时器

我们接着 8.7 节的内容继续介绍，其实通过使用 setInterval()函数定时功能，还可以实现时间计时器的功能。下面看一个具体的 JavaScript 代码实例。

【代码 8-8】（详见源代码目录 ch08-js-date-start-timer.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 日期和时间特效 - 时间计时器</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
    时间计时器:<span id="id-span-start-timer"></span>
    <button id="id-btn-start-timer" onclick="start_timer(this.id)">
                                                  开始计时器</button>
</div>
</body>
<script type="text/javascript">
    var iHours = 0;     // TODO: 定义小时的整数变量
    var iMinutes = 0;   // TODO: 定义分钟的整数变量
    var iSeconds = 0;   // TODO: 定义秒数的整数变量
    var strHours = "00";    // TODO: 定义小时的字符变量
    var strMinutes = "00";  // TODO: 定义分钟的字符变量
    var strSeconds = "00";  // TODO: 定义秒数的字符变量
    var strTimer;
    window.onload = function (ev) {
        // TODO: 定义计时器拼接的字符变量
        strTimer = strHours + ":" + strMinutes + ":" + strSeconds;
        document.getElementById('id-span-start-timer').
                                             innerHTML = strTimer;
    };
    function start_timer(thisid) {
        // TODO: 使用 setInterval 函数进行定时
```

```
33         window.setInterval(function () {
34             iSeconds++;
35             if(iSeconds==60){
36                 iSeconds = 0;
37                 iMinutes++;
38                 if(iMinutes==60){
39                     iMinutes = 0;
40                     iHours++;
41                 }
42             } else{
43                 strSeconds = iSeconds.toString();
44                 if(strSeconds.length == 1)
45                     strSeconds = "0" + strSeconds;
46                 strMinutes = iMinutes.toString();
47                 if(strMinutes.length == 1)
48                     strMinutes = "0" + strMinutes;
49                 strHours = iHours.toString();
50                 if(strHours.length == 1)
51                     strHours = "0" + strHours;
52             }
53             strTimer = strHours + ":" + strMinutes + ":" + strSeconds;
54             document.getElementById('id-span-start-timer').
                                                innerHTML = strTimer;
55         }, 1000);   // TODO: 间隔为 1 秒
56     }
57 </script>
58 </html>
```

关于【代码 8-8】的说明：

- 第 14 行代码通过<span>标签元素定义了一个行内容器，用于实现时间计时器的展示效果。
- 第 15 行代码通过<button>标签元素定义了一个按钮，添加了 onclick 事件处理方法（start_timer()），用户可以单击该按钮执行开始计时器的操作。
- 第 31 ~ 56 行代码是 start_timer()方法的实现过程，其中第 33 ~ 55 行代码通过调用 Window 对象的 setInterval()方法进行定时，时间间隔定义为 1 秒。每次定时（1 秒）完成后，通过第 19 ~ 24 行代码定义的变量（小时、分钟和秒数）进行累加操作，然后在第 53 行代码中将计时器整合成字符串格式保存在第 25 行代码定义的变量（strTimer）中，最后通过第 54 行代码在页面中进行显示。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 8.7 所示。

如图 8.7 中箭头所示，左边页面中为时间计时器初始状态，通过单击“开始计时器”操作后，右边页面中演示了时间计时器运行的效果（00:03:18）。

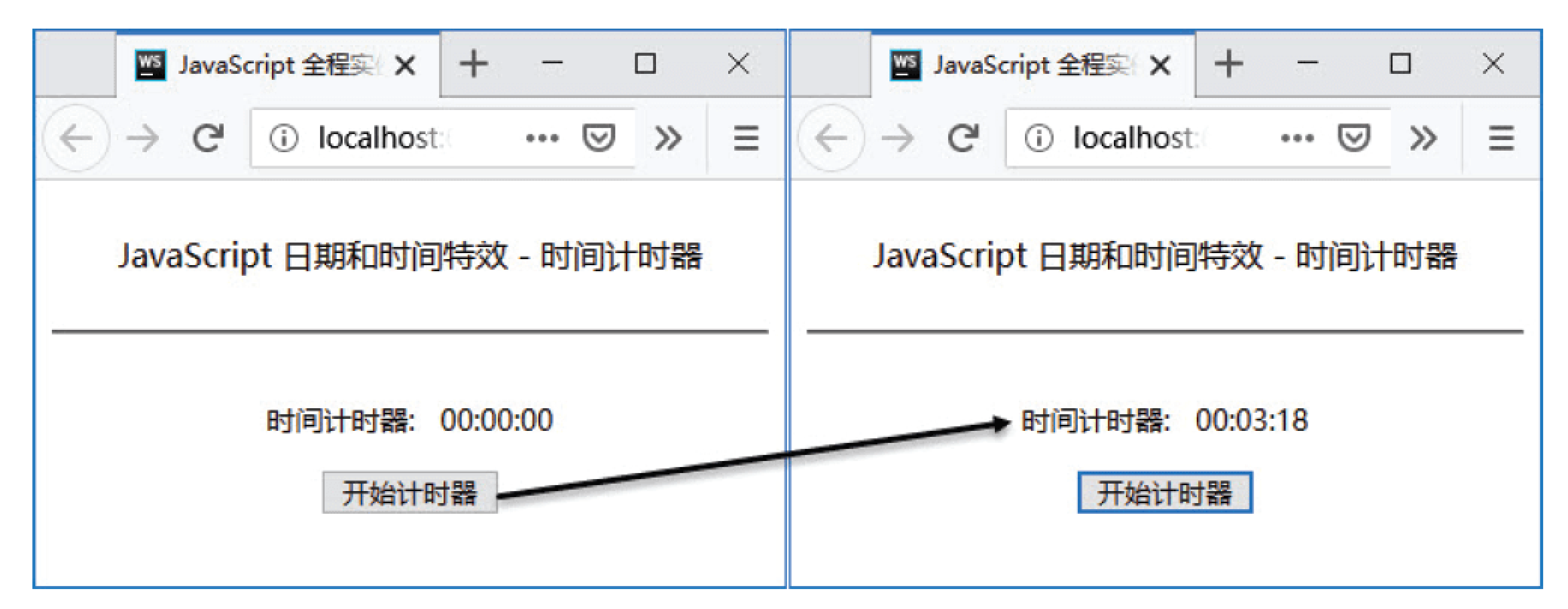


图 8.7　JavaScript 实现时间计时器

8.9　时间倒计时器

前一小节实现时间计时器的功能，接着我们逆向思维来实现时间倒计时器的功能。下面，看一个具体的 JavaScript 代码实例。

【代码 8-9】（详见源代码目录 ch08-js-date-count-down.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 日期和时间特效 - 时间倒计时器</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
    时间倒计时器:<span id="id-span-count-down"></span>
    起始时间:  <input type="text" id="id-input-init-time"
                            value="1">分钟
    <button id="id-btn-count-down" onclick="count_down(this.id)">开始倒计时
                                                              </button>
</div>
</body>
<script type="text/javascript">
```

```
var iHours = 0;     // TODO: 定义小时的整数变量
var iMinutes = 0;   // TODO: 定义分钟的整数变量
var iSeconds = 0;   // TODO: 定义秒数的整数变量
var strHours = "00";    // TODO: 定义小时的字符变量
var strMinutes = "00";  // TODO: 定义分钟的字符变量
var strSeconds = "00";  // TODO: 定义秒数的字符变量
var strTimer;
window.onload = function (ev) {
    init_time();
    // TODO: 定义计时器拼接的字符变量
    strTimer = strHours + ":" + strMinutes + ":" + strSeconds;
    document.getElementById('id-span-count-down').
                                          innerHTML = strTimer;
};
function init_time() {
    iMinutes = document.getElementById("id-input-init-time").value;
    strMinutes = iMinutes.toString();
    if (strMinutes.length == 1)
        strMinutes = "0" + strMinutes;
}
function count_down(thisid) {
    init_time();
    // TODO: 使用 setInterval 函数进行定时
    window.setInterval(function () {
        if (iSeconds == 0) {
            if (iMinutes > 0) {
                iMinutes--;
                strMinutes = iMinutes.toString();
                if (strMinutes.length == 1)
                    strMinutes = "0" + strMinutes;
                iSeconds = 60;
                iSeconds--;
                strSeconds = iSeconds.toString();
                if (strSeconds.length == 1)
                    strSeconds = "0" + strSeconds;
            } else if (iMinutes == 0) {
                strMinutes = "00";
            } else {
            }
        } else if (iSeconds > 0) {
            iSeconds--;
            strSeconds = iSeconds.toString();
            if (strSeconds.length == 1)
```

```
62                  strSeconds = "0" + strSeconds;
63            } else {
64            }
65            strTimer = strHours + ":" + strMinutes + ":" + strSeconds;
66            document.getElementById('id-span-count-down').
                                             innerHTML = strTimer;
67         }, 1000);   // TODO: 间隔为 1 秒
68      }
69  </script>
70  </html>
```

关于【代码 8-9】的说明：

- 第 14 行代码通过<span>标签元素定义了一个行内容器，用于实现时间倒计时器的展示效果。
- 第 15 行代码通过<input>标签元素定义了一个文本框，用于用户定义时间倒计时器的初始时间（默认值为 1 分钟）。
- 第 16 行代码通过<button>标签元素定义了一个按钮，添加了 onclick 事件处理方法（count_down()），用户可以单击该按钮执行开始倒计时器的操作。
- 第 33 ~ 38 行代码是 init_time()方法的实现过程，该方法用于获取用户输入的倒计时初始时间。
- 第 39 ~ 68 行代码是 count_down()方法的实现过程，其中第 42 ~ 67 行代码通过调用 Window 对象的 setInterval()方法进行定时，时间间隔定义为 1 秒。每次定时（1 秒）完成后，通过第 20 ~ 25 行代码定义的变量（小时、分钟和秒数）进行累减操作，然后在第 65 行代码中将计时器整合成字符串格式保存在第 26 行代码定义的变量（strTimer）中，最后通过第 66 行代码在页面中进行显示。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 8.8 和图 8.9 所示。

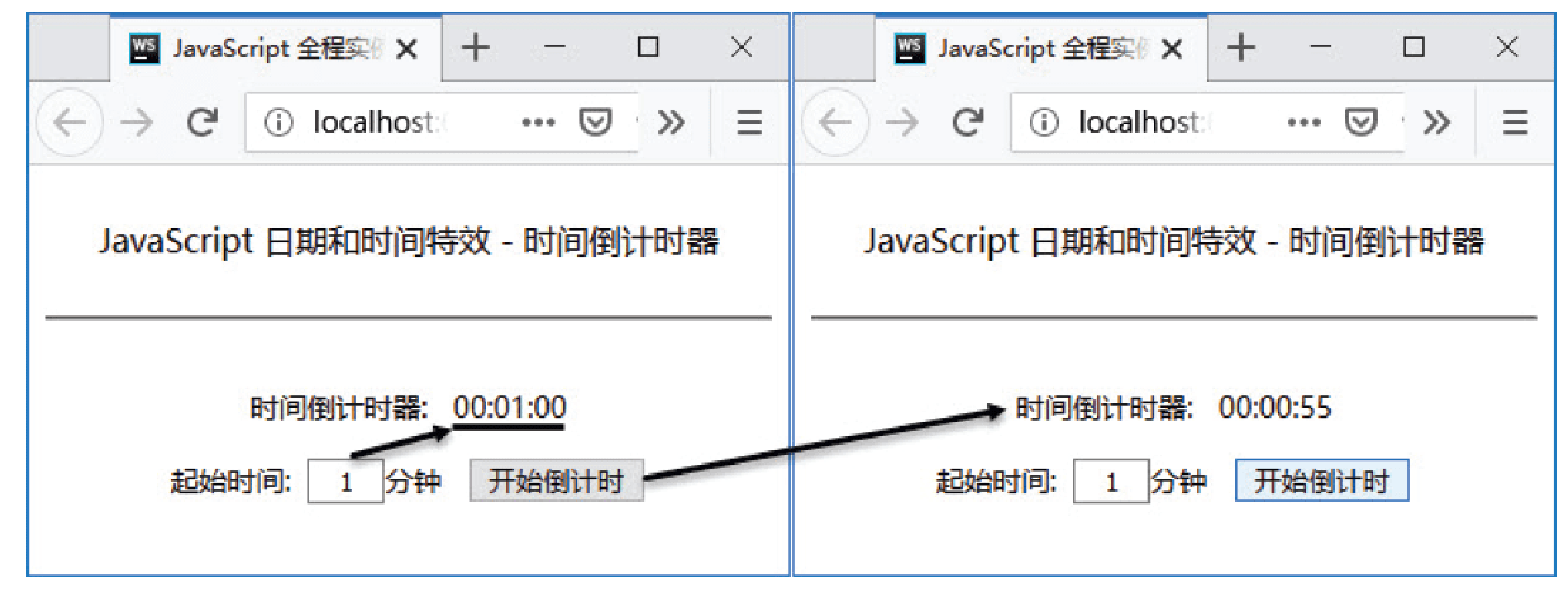


图 8.8　JavaScript 实现时间倒计时器（一）

如图 8.8 中箭头所示，左边页面中为时间倒计时器初始状态（00:01:00），通过单击“开始倒计时”按钮操作后，右边页面中演示了时间倒计时器运行的效果（00:00:55）。

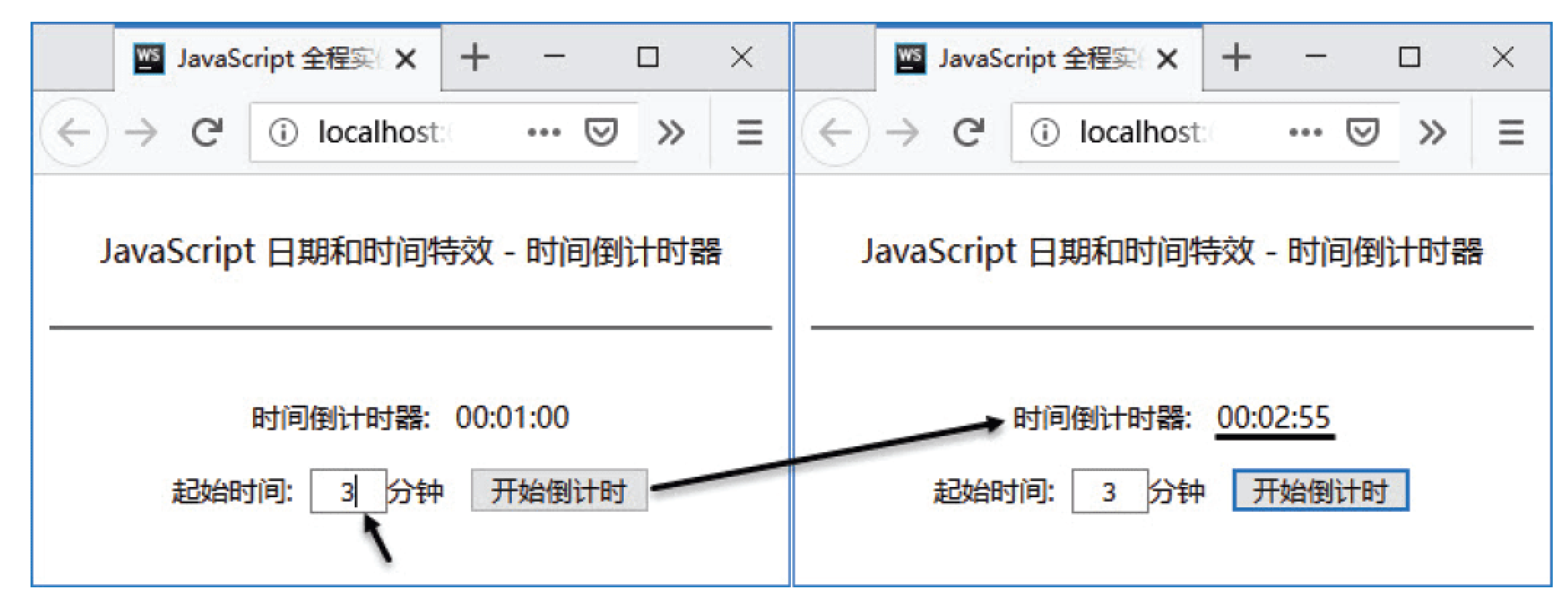


图 8.9　JavaScript 实现时间倒计时器（二）

如图 8.9 中箭头所示，左边页面中为时间倒计时器初始状态，用户手动输入了起始时间（3 分钟），通过单击“开始倒计时”按钮操作后，右边页面中演示了时间倒计时器运行的效果（00:02:55）。

8.10　计算时间差

在 JavaScript 脚本语言中，Date 对象提供了一个 getTime()方法，可以获取任一时间与 1970 年 1 月 1 日零点之间的时间差（毫秒单位）。利用 getTime()方法就可以很容易地计算出任意两个时间节点之间的时间差。

下面介绍一个通过 JavaScript 实现计算任意两个时间节点时间差的代码实例。

【代码 8-10】（详见源代码目录 ch08-js-date-cal-interval.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 日期和时间特效 - 计算时间差</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
    <label>第一个时间(2019-01-08 10:10:10):</label>
    <input type="text" name="time1" id="id-time-1"/>
    <label>第二个时间(2019-01-10 22:22:22):</label>
    <input type="text" name="time2" id="id-time-2"/>
```

```
18      <input type="button" value="计算" onclick="calTimeInterval();"/>
19      <span id="id-time-interval"></span>
20  </div>
21  </body>
22  <script type="text/javascript">
23      // TODO: calculate time interval
24      function calTimeInterval() {
25          var time1 = document.getElementById('id-time-1').value;
26          var time2 = document.getElementById('id-time-2').value;
27          var t1 = parseTime(time1);
28          var t2 = parseTime(time2);
29          var span = t1.getTime() - t2.getTime();
30          span = Math.abs(span / 1000);
31          document.getElementById("id-time-interval"). innerText =
                                                    '两个时间相差'+span+'秒.';
32      }
33      /**
34       * parse time
35       * @param str
36       * @returns {Date}
37       */
38      function parseTime(str) {
39          var date = str.split(' ')[0];
40          var darr = date.split('-');
41          var time = str.split(' ')[1];
42          var tarr = time.split(':');
43          var y = parseInt(darr[0]);
44          var m = parseInt(darr[1]);
45          var d = parseInt(darr[2]);
46          var h = parseInt(tarr[0]);
47          var mm = parseInt(tarr[1]);
48          var s = parseInt(tarr[2]);
49          return new Date(y, m, d, h, mm, s);
50      }
51  </script>
52  </html>
```

关于【代码 8-10】的说明：

- 第 15 行和第 17 行代码通过<input>标签元素定义了两个文本框，用于用户输入两个时间节点（注意格式为：YYYY-MM-DD hh:mm:ss）。
- 第 18 行代码通过<input>标签元素定义了一个按钮，添加了 onclick 事件处理方法（calTimeInterval()），用户可以单击该按钮执计算时间差的操作。

- 第 24 ~ 32 行代码是 calTimeInterval()方法的实现过程。其中，第 27 ~ 28 行代码通过调用自定义方法（parseTime()）来解析时间节点（返回 Date 类型）；第 29 行代码通过调用 getTime()方法获取这两个时间节点与 1970 年 1 月 1 日零点之间的时间差（毫秒单位）。
- 第 38 ~ 50 行代码是 parseTime()方法的实现过程，该方法负责将时间节点（格式：YYYY-MM-DD hh:mm:ss）解析为 Date 类型并返回。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 8.10 所示。

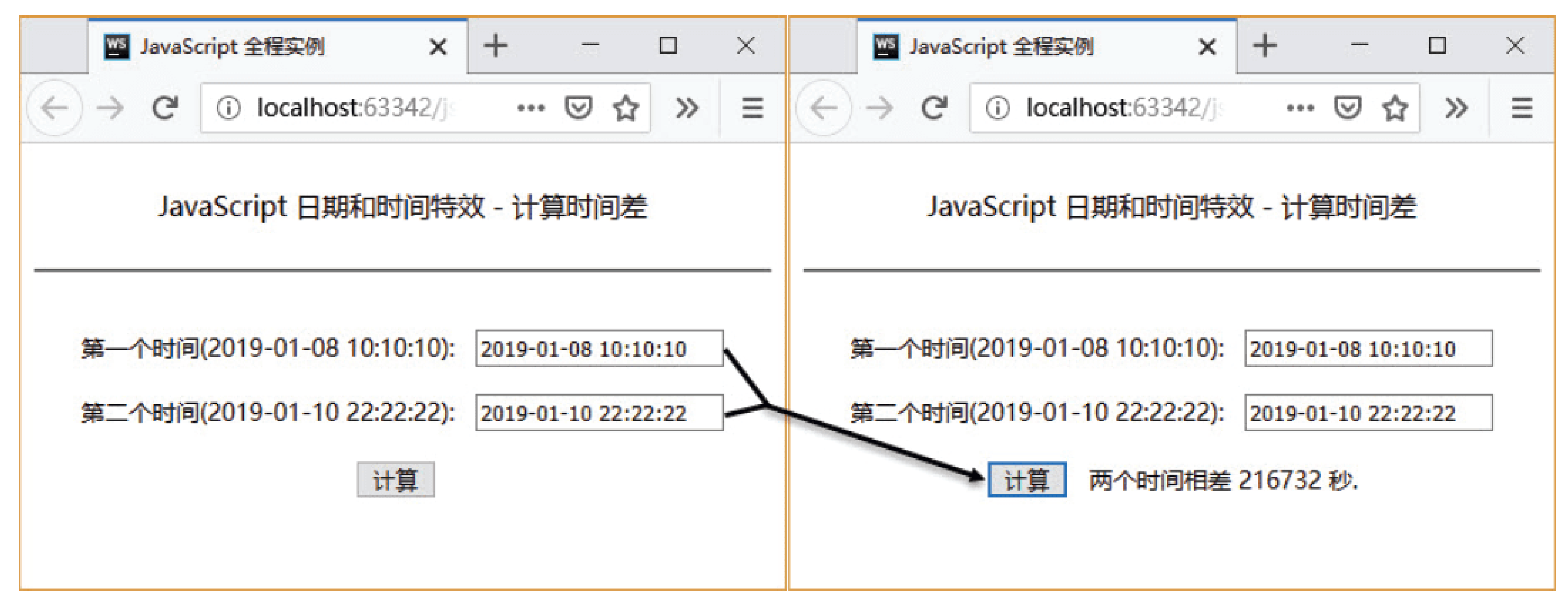


图 8.10　JavaScript 实现计算时间差

如图 8.10 中箭头所示，左边页面中填入了两个时间节点，通过单击“计算”按钮后，右边页面中显示了两个时间节点的时间差（216732 秒）。

8.11　计算日期间隔

在 8.10 节中，我们实现了计算任意两个时间节点之间的时间差功能。不过这个时间差的计算结果是以“秒”为单位的，可读性不是很好。在本节中，我们继续改进一下代码，直接计算出任意两个时间节点之间的日期间隔。下面看一个具体的 JavaScript 代码实例。

【代码 8-11】（详见源代码目录 ch08-js-date-cal-d2d.html 文件）

```
<script type="text/javascript">
    function calDateInterval() {
        var time1 = document.getElementById('id-time-1').value;
        var time2 = document.getElementById('id-time-2').value;
        var t1 = parseTime(time1);
        var t2 = parseTime(time2);
        var span = t1.getTime() - t2.getTime();
        span = Math.abs(span / 1000);
        var d2d = parseDate(span);
```

```
10         document.getElementById("id-date-interval").innerText =
                                                  '两个日期间隔: ' + d2d + ' .';
11     }
12     function parseTime(str) {
13         var date = str.split(' ')[0];
14         var darr = date.split('-');
15         var time = str.split(' ')[1];
16         var tarr = time.split(':');
17         var y = parseInt(darr[0]);
18         var m = parseInt(darr[1]);
19         var d = parseInt(darr[2]);
20         var h = parseInt(tarr[0]);
21         var mm = parseInt(tarr[1]);
22         var s = parseInt(tarr[2]);
23         return new Date(y, m, d, h, mm, s);
24     }
25     function parseDate(sec) {
26         var seconds = sec % 60;
27         var minutes = Math.floor(sec / 60);
28         var hours = Math.floor(minutes / 60);
29         var days = Math.floor(hours / 24);
30         var s = seconds;
31         var m = minutes - hours * 60;
32         var h = hours - days * 24;
33         var d = days;
34         return days + "天 " + h + "小时" + m + "分钟" + s + "秒";
35     }
36 </script>
```

关于【代码 8-11】的说明：

- 【代码 8-11】主要是在【代码 8-10】的基础上修改而成的，关键部分就是增加了一个自定义 parseDate()方法。
- 第 25 ~ 35 行代码是 parseDate()方法的实现过程，该方法负责将计算得出的时间间隔（秒）解析为“xx 天 xx 小时 xx 分钟xx 秒”的字符串格式并返回。这里主要是借助了 Math 对象的 floor()方法，将时间换算后进行了向下取整操作。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 8.11 所示。

如图 8.11 中箭头所示，左边页面中填入了两个时间节点，通过单击“计算”按钮后，右边页面中显示了两个日期的间隔（2 天 12 小时 12 分钟 12 秒）。

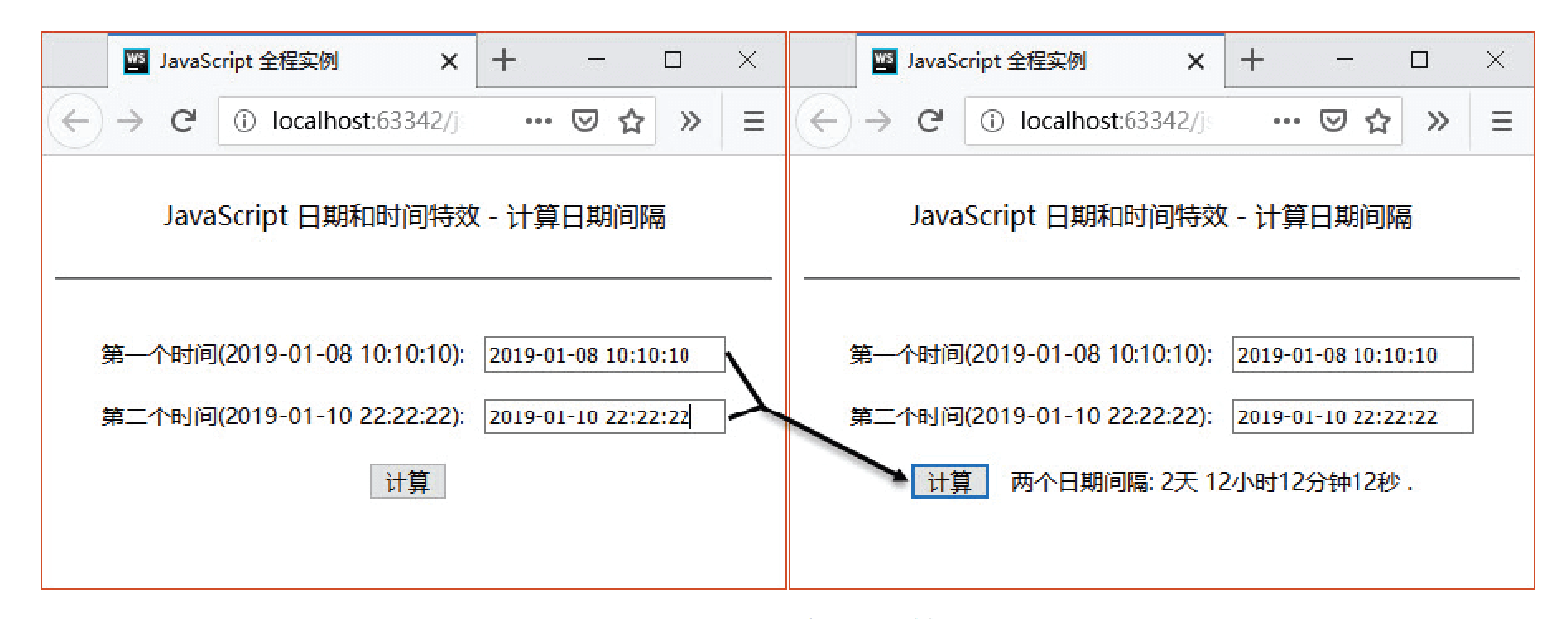


图 8.11　JavaScript 实现计算日期间隔

8.12　网页标题体现月进度

通常一个月分为上、中、下旬，合理安排好时间是月计划能够按照进度完成的有力保障。在本节中，我们实现一个网页标题能够体现月进度（上、中、下旬）变化的页面效果。下面看一下具体的 JavaScript 代码实例。

【代码 8-12】（详见源代码目录 ch08-js-date-month-title.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 日期和时间特效 - 网页标题体现月进度</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
    <span id="id-span-date"></span>
</div>
</body>
<script type="text/javascript">
    window.onload = function () {
        var strTitle;
        var date = new Date();
        var d = date.getDate();
        if(d <= 10) {
```

```
23                strTitle = '上旬，请安排好月计划';
24            } else if(d > 10 && d <= 20) {
25                strTitle = '中旬，请合理安排时间';
26            } else if(d > 20) {
27                strTitle = '下旬，请抓紧完成月计划';
28            } else {
29                strTitle = "";
30            }
31            document.title = strTitle;
32            var today = date.toLocaleDateString();
33            document.getElementById('id-span-date').innerText = "今天:" + today;
34        };
35    </script>
36    </html>
```

关于【代码 8-12】的说明：

- 第 21 行代码通过 getDate()方法获取了当前日期。
- 第 22～30 行代码通过 if 条件语句判断当前日期是月上旬、中旬或下旬，并根据判断结果定义不同的网页标题信息（strTitle）。
- 第 31 行代码通过将网页标题信息（strTitle）赋给 Document 对象的 title 属性来实现网页标题体现月进度的效果。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 8.12 所示。

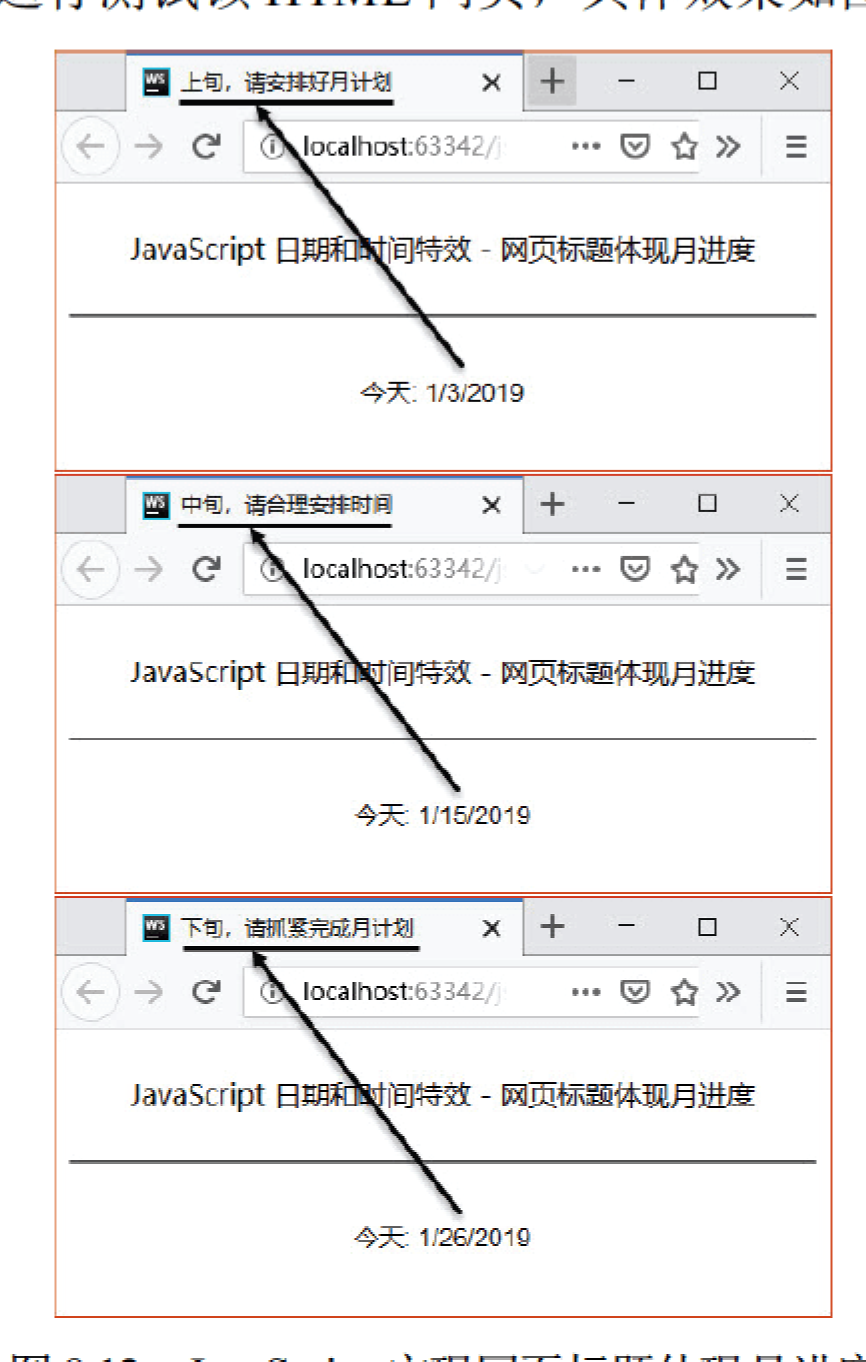


图 8.12　JavaScript 实现网页标题体现月进度

8.13　用表格制作日历

在 HTML 页面中，日期的最常用表现形式就是日历控件了。一般来讲，一年是 12 个月份，每个月份是 30 天左右，按照一周 7 天来排序，每个月份用一个 7 列的表格就可以很清晰的表示出来了。其实，常规的 HTML 日历控件就可以使用表格<table>元素，再结合 JavaScript 脚本语言动态编程制作出来。

下面，具体介绍一个通过 JavaScript 实现用表格制作日历的代码实例。

【代码 8-13】（详见源代码目录 ch08-js-date-tb-calendar.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 日期和时间特效 - 用表格制作日历</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center">
    <p>
        点击选择日历:
        <select id="y"></select>年
        <select id="m"></select>月
    </p>
    <table id="id-tb-date" border="1" align="center"></table>
    <span id="id-span-info"></span>
</div>
</body>
<script type="text/javascript">
    window.onload = function () {
        var y = document.getElementById('y');
        var m = document.getElementById('m');
        var strY = '';
        for (let i = 2019; i <= 2020; i++) {
            strY += '<option value="' + i + '">' + i + '</option>';
        }
        y.innerHTML = strY;
```

```
        strM = '';
        for (let j = 1; j <= 12; j++) {
            strM += '<option value="' + j + '">' + j + '</option>';
        }
        m.innerHTML = strM;
        m.onchange = function () {
            var year = document.getElementById('y').value;
            year = parseInt(year);
            var month = parseInt(this.value);
            var days = 31;
            if (isRunYear(year) && month == 2) {
                days = 29;
            } else if (!isRunYear(year) && month == 2) {
                days = 28;
            } else if (month == 4
                || month == 6
                || month == 9
                || month == 11) {
                days = 30;
            }
            var arrWeek = ['日', '一', '二', '三', '四', '五', '六'];
            var strW = '<tr>';
            for (var w = 0; w < 7; w++) {
                strW += '<td>星期' + arrWeek[w] + '</td>';
            }
            strW += '</tr>';
            var date = new Date(year, month - 1, 1);
            var week = date.getDay();
            if (week == 7)
                week = 0;
            var j = 1;
            while (true) {
                strW += '<tr>';
                for (var i = 0; i < 7; i++) {
                    if (j == 1 && i == week) {
                        strW += '<td onclick="showDay(this,'+j+','+i+');">1</td>';
                        j++;
                    } else if (j > 1 && j <= days) {
                        strW += '<td onclick="showDay(this,'+j+','+i+');">'+j+'</td>';
                        j++;
                    } else
```

```
073                         strW += '<td> </td>';
074                     }
075                     strW += '</tr>';
076                     if (j > days)
077                         break;
078                 }
079                 document.getElementById('id-tb-date').innerHTML = strW;
080             };
081         };
082         function isRunYear(y) {
083             return y % 4 == 0;
084         }
085         function showDay(el, d, w) {
086             console.log(el);
087             setTbBgColor(el);
088             var y = document.getElementById('y').value;
089             var m = document.getElementById('m').value;
090             var arrW = ['日', '一', '二', '三', '四', '五', '六'];
091             var strW = '星期';
092             for (let i = 0; i < 7; i++) {
093                 if (i == w) {
094                     strW += arrW[i];
095                     break;
096                 }
097             }
098             var strInfo = '您选择的日期是：' + y + '年' + m + '月' + d + '日，
                                                                ' + strW + '.';
099             document.getElementById('id-span-info').innerText = strInfo;
100         }
101         function setTbBgColor(el) {
102             var tds = document.getElementsByTagName("td");
103             for (let i = 0; i < tds.length; i++) {
104                 tds[i].style.backgroundColor = "";
105             }
106             el.style.backgroundColor = "gray";
107         }
108     </script>
109 </html>
```

关于【代码 8-13】的说明：

- 第 016 行和第 017 行代码分别使用<selelct>标签元素定义了两个下拉列表框，用于显示提供给用户进行选择的年份（<selelct id="y">）和月份（<selelct id="m">）。

- 第 019 行代码使用<table id="id-tb-date">标签元素定义了一个表格，用于显示通过 JavaScript 脚本动态生成的日历。
- 第 020 行代码使用<span id="id-span-info">标签元素定义了一个行内容器，用于显示用户通过单击所选择具体日期的提示信息。
- 第 024 ~ 081 行代码定义了页面初始化加载（onload）的过程代码。其中，第 028 ~ 030 行代码和第 033 ~ 035 行代码通过使用 for 循环语句初始化了年份（2019 ~ 2020）和月份（1 ~ 12），用户可以通过选择具体的年份和月份来动态创建并显示表格日历。第 037 ~ 080 行代码定义了月份（<selelct id="m">）下拉列表框的 onchange 事件处理方法。第 042 ~ 051 行代码用于筛选月份（大月、小月和闰月）的天数（31 天、30 天、28 天或 29 天），其中自定义 isRunYear() 方法用于判断闰年（闰月为 29 天）。第 052 ~ 057 行代码用于动态生成表格日历的星期栏（星期日 ~ 星期六，符合西历标准）。第 058 ~ 079 行代码用于动态生成具体的日历，其中第 067 行代码和第 070 行代码定义了<td>标签元素的单击 onclick 事处理方法（showDay()），当用户单击具体日期时会在页面中显示关于日期的提示信息。
- 第 085 ~ 100 行代码是自定义 showDay()方法的实现过程，该方法会将用户单击选择的日期（以“年、月、日 星期几”的格式）在<span id="id-span-info">标签元素中进行显示。
- 第 101 ~ 107 行代码是自定义 setTbBgColor()方法的实现过程，该方法会突出显示表格日历中用户单击选择的日期的背景色。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 8.13 所示。

图 8.13　JavaScript 实现用表格制作日历

如图 8.13 中箭头所示，左边页面中显示了表格日历（2019 年 1 月）动态生成后的效果，右边页面中显示了用户选择具体日期（2019 年 1 月 2 日）后的效果。

8.14　日期输入框

在 HTML 5 语法中，专门新增了一个日期输入框控件，使用<input type="date">标签就可以实

现。那么，之前 HTML 版本中的日期输入框控件是如何实现的呢？其实，基于 8.13 节中实现的表格日历，再结合使用文本框，同样可以通过 JavaScript 脚本语言动态生成日期输入框控件。下面具体介绍一个通过 JavaScript 实现日期输入框的代码实例。

【代码 8-14】（详见源代码目录 ch08-js-date-input-calendar.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 日期和时间特效 - 日期输入框</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
    <p>
        日期输入框:<input type="text" id="id-input-date"
                                            onfocus="initDate(this.id);"/>
    </p>
    <div id="id-div-date" style="">
        <p>
            <select id="y"></select>年
            <select id="m" onchange="fillTbCalendar();"></select>月
        </p>
        <table id="id-tb-date" border="1" align="center"></table>
    </div>
</div>
</body>
<script type="text/javascript">
    window.onload = function () {
        document.getElementById("id-input-date").value = "";
        fillSelCalendar();
    };
    function fillSelCalendar() {
        var y = document.getElementById('y');
        var m = document.getElementById('m');
        var strY = '';
        for (let i = 2019; i <= 2020; i++) {
```

```
            strY += '<option value="' + i + '">' + i + '</option>';
        }
        y.innerHTML = strY;
        strM = '';
        for (let j = 1; j <= 12; j++) {
            strM += '<option value="' + j + '">' + j + '</option>';
        }
        m.innerHTML = strM;
    }
    function initDate(thisid) {
        document.getElementById(thisid).value = "";
        var now = new Date();
        var yyyy = now.getFullYear();
        var mm = now.getMonth() + 1;
        var dd = now.getDate();
        var week = now.getDay();
        var arrWeek = ['日', '一', '二', '三', '四', '五', '六'];
        var strWeek = '星期';
        for (let i = 0; i < 7; i++) {
            if (i == week) {
                strWeek += arrWeek[i];
                break;
            }
        }
        var strInfo = yyyy + '年' + mm + '月' + dd + '日 ' + strWeek;
        document.getElementById(thisid).value = strInfo;
        var divDate = document.getElementById('id-div-date');
        var year = document.getElementById('y');
        var month = document.getElementById('m');
        year.value = yyyy;
        month.value = mm;
        divDate.style.display = 'block';
        fillTbCalendar();
    }
    function fillTbCalendar(){
        var year = document.getElementById('y').value;
        year = parseInt(year);
        var month = parseInt(document.getElementById('m').value);
        var days = 31;
        if (isRunYear(year) && month == 2) {
            days = 29;
```

```
        } else if (!isRunYear(year) && month == 2) {
            days = 28;
        } else if (month == 4
            || month == 6
            || month == 9
            || month == 11) {
            days = 30;
        }
        var arrWeek = ['日', '一', '二', '三', '四', '五', '六'];
        var strW = '<tr>';
        for (var w = 0; w < 7; w++) {
            strW += '<td>星期' + arrWeek[w] + '</td>';
        }
        strW += '</tr>';
        var date = new Date(year, month - 1, 1);
        var week = date.getDay();
        if (week == 7)
            week = 0;
        var j = 1;
        while (true) {
            strW += '<tr>';
            for (var i = 0; i < 7; i++) {
                if (j == 1 && i == week) {
                    strW += '<td onclick="showDay(this,' + j + ',
                                                       ' + i + ');">1</td>';
                    j++;
                } else if (j > 1 && j <= days) {
                    strW += '<td onclick="showDay(this,' + j + ',' + i + ');
                                                       ">' + j + '</td>';
                    j++;
                } else
                    strW += '<td> </td>';
            }
            strW += '</tr>';
            if (j > days)
                break;
        }
        document.getElementById('id-tb-date').innerHTML = strW;
    }
    function isRunYear(y) {
        return y % 4 == 0;
    }
    function showDay(el, d, w) {
```

```
118         setTbBgColor(el);
119         var y = document.getElementById('y').value;
120         var m = document.getElementById('m').value;
121         var arrW = ['日', '一', '二', '三', '四', '五', '六'];
122         var strW = '星期';
123         for (let i = 0; i < 7; i++) {
124             if (i == w) {
125                 strW += arrW[i];
126                 break;
127             }
128         }
129         var strInfo = y + '年' + m + '月' + d + '日 ' + strW;
130         document.getElementById('id-input-date').value = strInfo;
131     }
132     function setTbBgColor(el) {
133         var tds = document.getElementsByTagName("td");
134         for (let i = 0; i < tds.length; i++) {
135             tds[i].style.backgroundColor = "";
136         }
137         el.style.backgroundColor = "gray";
138     }
139 </script>
140 </html>
```

关于【代码 8-14】的说明：

- 【代码 8-14】是在【代码 8-13】的基础上改进而成的，主要是第 015 行代码增加了一个日期输入框，用于显示用户选择的日期（初始状态为当前日期）。
- 第 045 ~ 069 行代码实现的自定义 initDate()方法用于初始化日期输入框，日期输入框的初始化状态为当前日期。
- 第 070 ~ 113 行代码实现的自定义 fillTbCalendar()方法用于初始化表格日历，初始状态为当前日期的月份日历。
- 第 117 ~ 131 行代码实现的自定义 showDay()方法用于在日期输入框中显示用户选择的日期，日期格式为“YYYY 年 MM 月 DD 日 星期 W”。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 8.14 所示。

如图 8.14 中箭头所示，左边页面中显示了初始状态（当前日期）的日期输入框，右边页面中显示了用户选择（2019 年 1 月 8 日 星期二）的日期输入框。

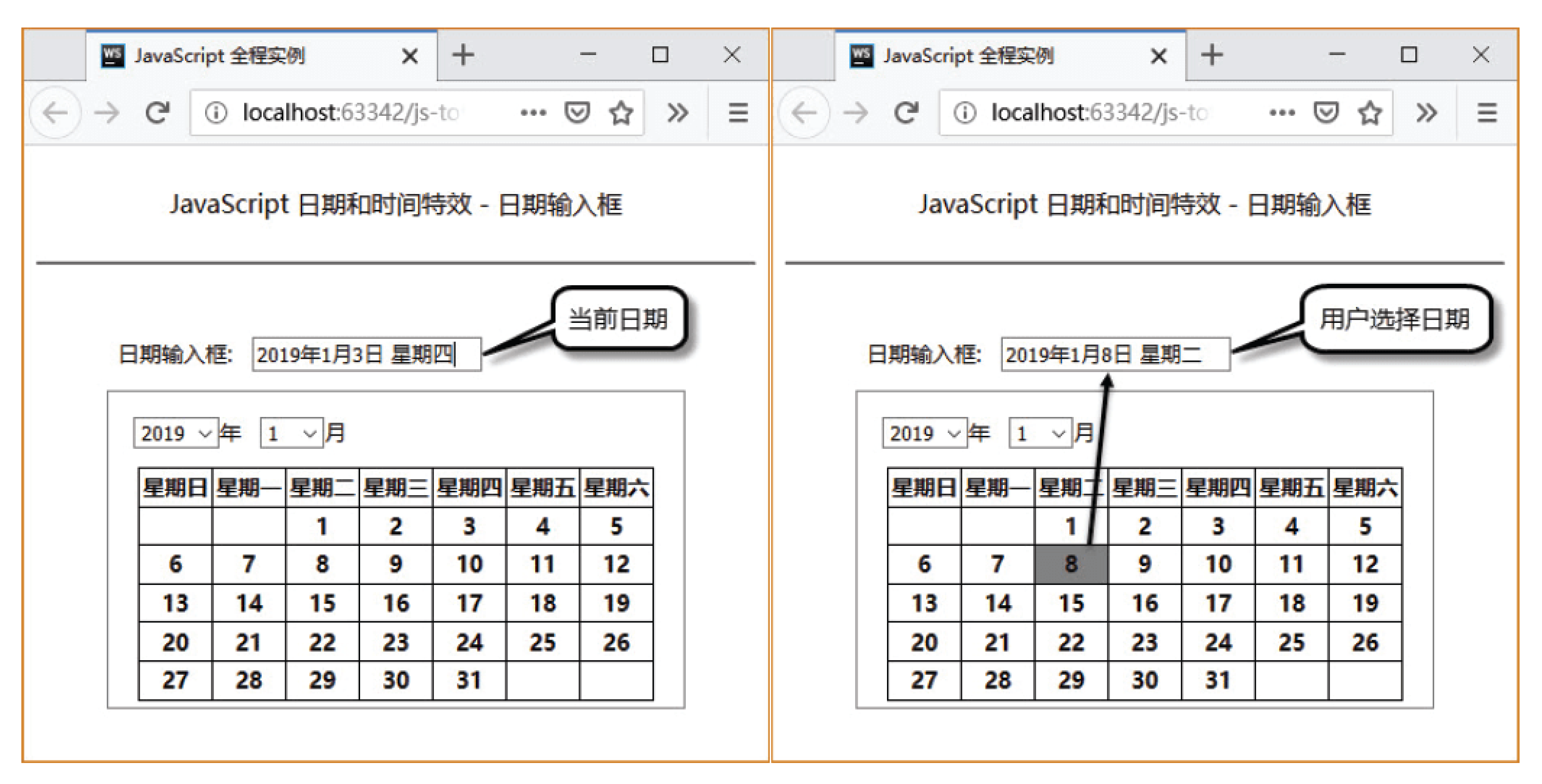


图 8.14　JavaScript 实现日期输入框

8.15　显示网页登录时间

很多网站会有一个显示用户登录页面时间的功能，基本原理都是通过设定计时器来实现的。下面看一个通过 JavaScript 实现显示网页登录时间的代码实例。

【代码 8-15】（详见源代码目录 ch08-js-date-login-time.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 日期和时间特效 - 显示网页登录时间</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
    <span id="id-span-login-time"></span>
</div>
</body>
<script type="text/javascript">
    var s = 0;  // TODO: 秒
    var m = 0; // TODO: 分钟
```

```
20      var h = 0; // TODO: 小时
21      window.onload = function () {
22          window.setInterval(function () {
23              s++;    // TODO: 秒+1
24              if (s == 60) {  // TODO: 如果秒为 60
25                  s = 0;  // TODO: 秒置零
26                  m += 1; // TODO: 分钟+1
27              }
28              if (m == 60) {  // TODO: 如果分钟为 60
29                  m = 0;  // TODO: 分钟置零
30                  h += 1; // TODO: 小时+1
31              }
32              var strLoginTime = "您已登录,累计时间:" + h + " 时 " + m +
                                                " 分 " + s + " 秒.";
33              document.getElementById('id-span-login-time').innerText =
                                                strLoginTime;
34          }, 1000);
35      };
36  </script>
37  </html>
```

关于【代码 8-15】的说明：

- 第 18 ~ 20 行代码定义了一组变量（s、m、h），分别用于表示秒、分钟和小时。
- 第 22 ~ 34 行代码通过 setInterval()方法定义了一个计时器，时间间隔为 1000 毫秒（1 秒）。在每一次计时器定义的时间间隔（1 秒）内，对秒数变量（s）进行一次累加（+1）操作；而当变量（s）累加到 60（1 分钟）时，对分钟变量（m）进行一次累加（+1）操作；而当变量（m）累加到 60（1 小时）时，再对小时变量（h）进行一次累加（+1）操作。按照这个方法进行时间的累加，从而实现对用户登录网页时间的计时功能。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 8.15 所示。

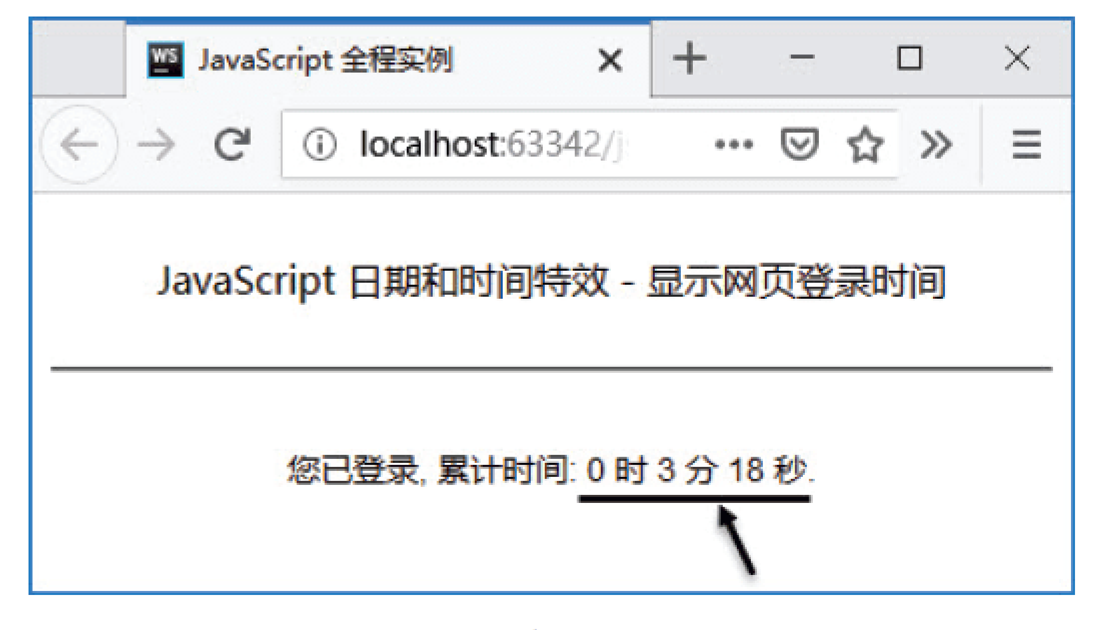


图 8.15　JavaScript 实现显示网页登录时间

如图 8.15 中箭头所示，页面中显示了用户登录网页的累计时间（0 时 3 分 18 秒）。

第 9 章　网 页 特 效

本章介绍如何通过 JavaScript 来实现网页的各种特效，通过这些特效来丰富 HTML 页面的设计手段和展示效果。

9.1　网 页 概 述

网页是构成网站的主体元素，一个网页是由各种 HTML 标签和各类资源所构成的超文本文件。

在 JavaScript 脚本语言中，除了可以针对各种页面元素进行编程设计外，还可以针对整体网页（浏览器特性、窗口属性、滚动条控制、键盘鼠标控制和页面特效等）进行编程设计。

JavaScript 网页特效是网站开发中很重要的一个环节，很多很特殊的功能都是通过网页特效来实现的。本章将为读者介绍多种网页特效的代码实例。

9.2　打开新页面

在 HTML 页面中打开新页面可以使用很多种方法来实现，比如直接使用超链接<a>标签元素，或者使用 Window 对象的 open()方法，又或者直接使用“javascript:open”脚本语言的方式。在本节中，我们就详细介绍几种打开新页面的代码实例。

【代码 9-1】（详见源代码目录 ch09-js-html-open.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 网页特效 - 打开新页面</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
```

```
14      <a href="open.html">打开新页面</a>
15      <a href="open.html" target="_blank">打开新页面(target="_blank")</a>
16      <a onclick="window.open('open.html')">打开新页面(window.open)</a>
17      <input type="button" value="打开一个新的窗口(window.open)"
18            onclick="window.open('open.html')"/>
19      <input type="button" value="打开一个新的窗口(function)"
20            onclick="openNewHtml('open.html')"/>
21      <a href="javascript:openNewHtml('open.html')">打开新页面(javascript:function)</a>
22  </div>
23  </body>
24  <script type="text/javascript">
25      function openNewHtml(url) {
26          window.open(url);
27      }
28  </script>
29  </html>
```

关于【代码 9-1】的说明：

- 第 14 行和第 15 行代码分别通过<a>标签元素定义了一个超链接，href 属性值定义了一个新的网页（href="open.html"）。这两行代码均可以打开新页面，但区别是第 15 行代码定义的<a>标签元素增加定义了一个 target 属性（target="_blank"），表示新页面将会在新窗口中打开。具体的页面效果如图 9.1 和图 9.2 所示。
- 第 16 行代码通过<a>标签元素定义了一个超链接，不过是使用 onclick 事件方法取代了 href 属性，在 onclick 事件方法中直接调用 Window 对象的 open()方法打开新页面（该方法默认是在新窗口中打开网页）。
- 第 17～18 行代码和第 19～20 行代码分别通过<input type="button">标签元素定义了一个按钮，然后通过 onclick 事件方法实现打开新页面的操作。二者的区别是，第 20 行代码定义的 onclick 事件方法是通过调用一个自定义 openNewHtml()方法来实现的。第 25～27 行代码是 openNewHtml()方法的实现过程，不过第 26 行代码仍旧是通过调用 Window 对象的 open()方法实现打开新页面的。
- 第 21 行代码定义的<a>标签元素比较特殊，在 href 属性值中是直接通过定义 JavaScript 脚本（javascript:openNewHtml()）的方式打开新页面的，这种使用方式虽不常用，但也需要读者理解掌握。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 9.1 和图 9.2 所示。

如图 9.1 中箭头和标识所示，在使用第 14 行代码定义的超链接打开新页面时，是在当前窗口中打开的。

如图 9.2 中箭头和标识所示，在使用第 15 行代码定义的超链接打开新页面时，是在新窗口中打开的。

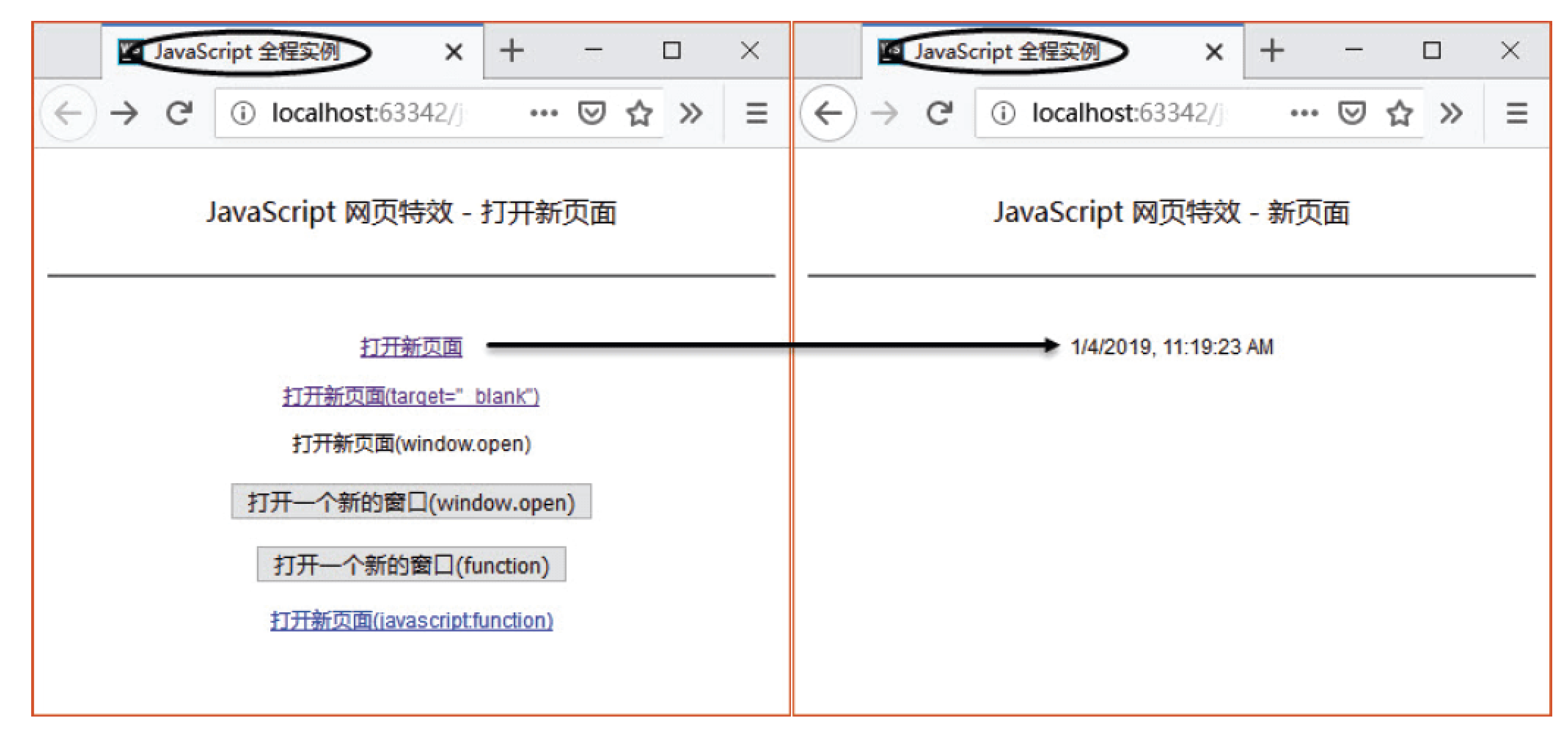


图 9.1　JavaScript 实现打开新页面（一）

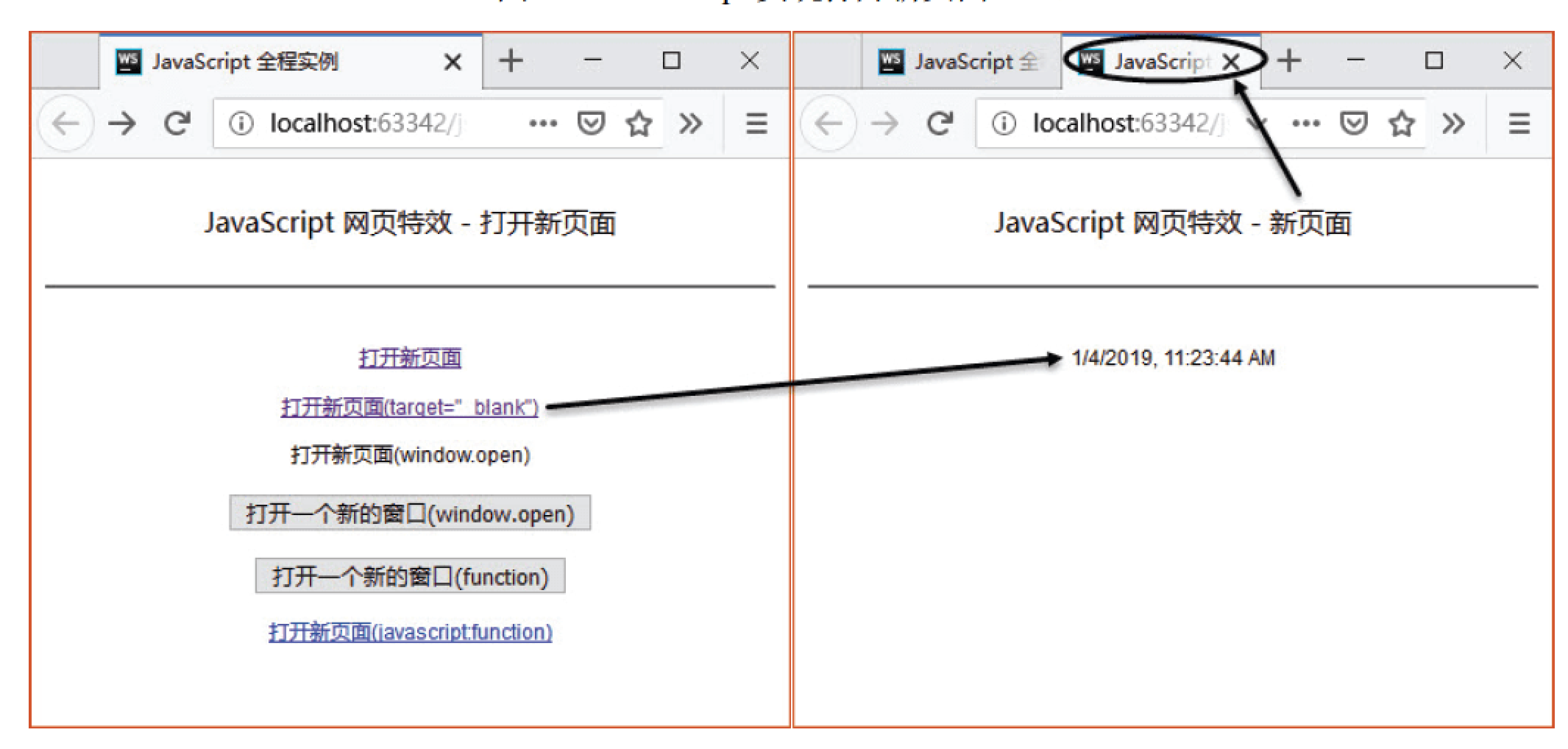


图 9.2　JavaScript 实现打开新页面（二）

9.3　打开指定大小的窗口

在 9.2 节中，我们初步介绍了使用 Window 对象的 open()方法打开新窗口。其实，通过 open()方法还可以自定义新窗口的特性，最常用的就是指定新窗口的大小尺寸。在本节中将详细介绍一个打开指定大小窗口的 JavaScript 代码实例。

【代码 9-2】（详见源代码目录 ch09-js-html-open-size.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
```

```
05     <title>JavaScript 全程实例</title>
06  </head>
07  <body>
08  <!-- 添加文档主体内容 -->
09  <header>
10     <nav>JavaScript 网页特效 - 打开指定大小的窗口</nav>
11  </header>
12  <!-- 添加文档主体内容 -->
13  <div id="id-div-center" style="">
14     指定新窗口大小:
15     Width:<input type="text" id="id-window-width" value="">
16     Height:<input type="text" id="id-window-height" value="">
17     <input type="button" value="打开指定大小的窗口"
                                    onclick="openNewHtml('open.html')"/>
18  </div>
19  </body>
20  <script type="text/javascript">
21     function openNewHtml(url) {
22        var w, h;
23        w = document.getElementById('id-window-width').value;
24        if(w == 0)
25           w = 100;
26        h = document.getElementById('id-window-height').value;
27        if(h == 0)
28           h = 100;
29        window.open(url, '', 'width=' + w + ',height=' + h);
30     }
31  </script>
32  </html>
```

关于【代码 9-2】的说明：

- 第 15 行和第 16 行代码通过<input>标签元素定义了一组文本框，用于接收用户自定义的窗口大小尺寸（宽度和高度）。
- 第 17 行代码通过<input type="button">标签元素定义了一个按钮，并添加了单击 onclick 事件处理方法（openNewHtml()），用于打开新窗口页面。
- 第 21～30 行代码是 openNewHtml()方法的实现过程。其中，第 29 行代码使用 Window 对象的 open()方法打开指定宽度和高度的新窗口。另外，第 24～28 行代码的作用是，如果用户未指定新窗口的宽度和高度，则指定默认值为 100。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 9.3 所示。

如图 9.3 中的箭头所示，新打开的窗口尺寸为用户设定的宽度值（300）和高度值（150）。

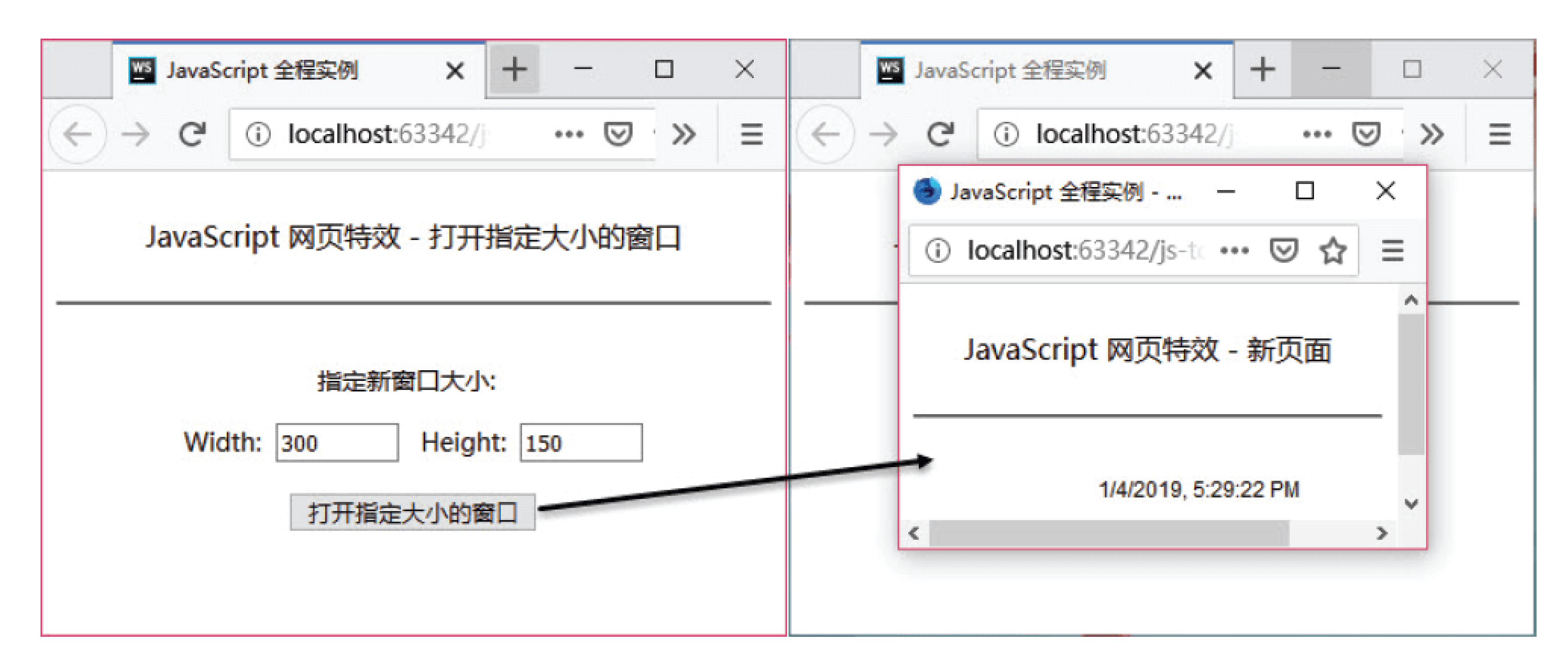


图 9.3　JavaScript 实现打开指定大小的窗口

9.4　获取打开子窗口的父窗口

在 9.3 节中，我们介绍了如何打开子窗口页面。试想一下，如果网站中的父窗口和子窗口很多、关系比较复杂，那么如何厘清窗口之间的父子关系呢？这里就需要用到 Window 对象中一个非常有用的 opener 属性了，该属性表示创建当前子窗口的 Window 对象引用（乍听起来比较拗口，其实就是指的父窗口）。

在本节中，简单介绍一个获取打开子窗口的父窗口的 JavaScript 代码实例。这里，我们需要创建两个页面文件，一个是父窗口页面，另一个是子窗口页面。首先，看一下父窗口的页面代码。

【代码 9-3】（详见源代码目录 ch09-js-html-parent.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 网页特效 - 获取打开子窗口的父窗口</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
<input type="button" value="打开子窗口"onclick="openNewWindow
                                        ('ch09-js-html-sub.html')"/>
    <span id="id-span-parent"></span>
</div>
```

```
17 </body>
18 <script type="text/javascript">
19     function openNewWindow(url) {
20         var curUrl = window.location.href.split('?')[0];
21         document.getElementById('id-span-parent').innerHTML +=
                                               "Current url: " + curUrl;
22         var subWin = window.open(url);
23         var winUrl = subWin.opener.location.href.split('?')[0];
24         document.getElementById('id-span-parent').innerHTML +=
                                               "Return url: " + winUrl;
25     }
26 </script>
27 </html>
```

关于【代码 9-3】的说明：

- 第 14 行代码通过<input>标签元素定义了一个按钮，并定义了单击 onclick 事件处理方法（openNewWindow()），用于打开子窗口。
- 第 19～25 行代码是 openNewWindow()方法的实现过程。首先，第 20 行代码使用 Location 对象的 href 属性获取了当前页面（父页面窗口）地址。然后，第 22 行代码通过 Window 对象的 open 方法打开了新页面（子页面窗口），该方法的返回值保存在变量（subWin）中。最后，第 23 行代码通过变量（subWin）调用 opener 属性，获取了变量（subWin）所代表窗口的地址。

然后，看一下子窗口的页面代码。

【代码 9-4】（详见源代码目录 ch09-js-html-sub.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 网页特效 - 子窗口页面</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
    <span id="id-span-sub"></span>
</div>
</body>
<script type="text/javascript">
```

```
18      window.onload = function () {
19          var curUrl = window.location.href.split('?')[0];
20          document.getElementById('id-span-sub').innerHTML +=
                                                    "Current url: " + curUrl;
21          var openerHref = window.opener.location.href.split('?')[0];
22          document.getElementById('id-span-sub').innerHTML +=
                                                    "opener url: " + openerHref;
23      };
24  </script>
25  </html>
```

关于【代码 9-4】的说明：

- 第 19 行代码使用 Location 对象的 href 属性获取了当前页面（子页面窗口）地址。
- 第 21 行代码通过 Window 对象的 opener 属性，获取父窗口页面的地址。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 9.4 所示。



图 9.4　JavaScript 实现获取打开子窗口的父窗口

如图 9.4 中的箭头和标识所示，在新打开的子窗口中，通过 opener 属性成功获取了父窗口的地址。

9.5 父子窗口之间数据交互

在 9.4 节中，我们初步介绍了使用 opener 属性获取父窗口地址的方法。其实，通过 opener 属性还可以实现父窗口和子窗口之间的数据交互。在本节中，介绍一个通过 JavaScript 实现父子窗口之间数据交互的代码实例。

首先，看一下父窗口的页面代码。

【代码 9-5】（详见源代码目录 ch09-js-html-parent-sdata.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 网页特效 - 父窗口传递数据</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
    <label for="id-input-user">用户数据：</label><input type="text"
            id="id-input-user">
    <input value="打开子窗口" onclick="openSubWindow
                                        ('ch09-js-html-sub-sdata.html')"/>
    <span id="id-span-parent"></span>
</div>
</body>
<script type="text/javascript">
    function openSubWindow(url) {
        var subWin = window.open(url);
    }
</script>
</html>
```

关于【代码 9-5】的说明：

- 第 14 行代码通过<input id="id-input-user">标签元素定义了一个文本框，用户可以输入一个随意数据。该文本框既可以将数据传递给子窗口页面，也可以用来接收来自子窗口页面的数据。
- 第 15 行代码通过<input>标签元素定义了一个按钮，并定义了单击 onclick 事件处理方法（openSubWindow()），用于打开子窗口。
- 第 20～22 行代码是 openSubWindow()方法的实现过程。其中，第 21 行代码通过 Window 对象的 open 方法打开了新页面（子页面窗口）。

然后，看一下子窗口的页面代码。

【代码 9-6】（详见源代码目录 ch09-js-html-sub-sdata.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 网页特效 - 子窗口传递数据</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
    <span id="id-span-sub"></span>
    <label for="id-input-return">回传数据:</label><input type="text"
            id="id-input-return">
    <input type="button" value="回传数据"
           onclick="returnData('id-input-return')"/>
</div>
</body>
<script type="text/javascript">
    window.onload = function () {
        var openerVal = window.opener.document.getElementById
                                  ('id-input-user').value;
        document.getElementById('id-span-sub').innerHTML+=
                                  "opener value:"+openerVal;
    };
    function returnData(id) {
        var vReturnData = document.getElementById(id).value;
```

```
26          window.opener.document.getElementById('id-input-user').
                                                    value = vReturnData;
27      }
28  </script>
29  </html>
```

关于【代码 9-6】的说明：

- 第 21 行代码通过 Window 对象的 opener 属性获取了父窗口页面中文本框中的数据，也就是【代码 9-5】中第 14 行代码定义的文本框（<input id="id-input-user">）。
- 第 15 行代码通过<input id="id-input-return">标签元素定义了一个文本框，用户可以输入一个随意数据，该数据用于回传到父窗口页面中。
- 第 16 行代码通过<input>标签元素定义了一个按钮，并定义了单击 onclick 事件处理方法（returnData()），用于执行回传数据的操作。
- 第 24～27 行代码是 returnData()方法的实现过程。其中，第 26 行代码通过 Window 对象的 opener 属性将第 15 行代码中用户输入的数据回传到父窗口页面，也就是回传到【代码 9-5】中第 14 行代码定义的文本框（<input id="id-input-user">）中。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 9.5 所示。

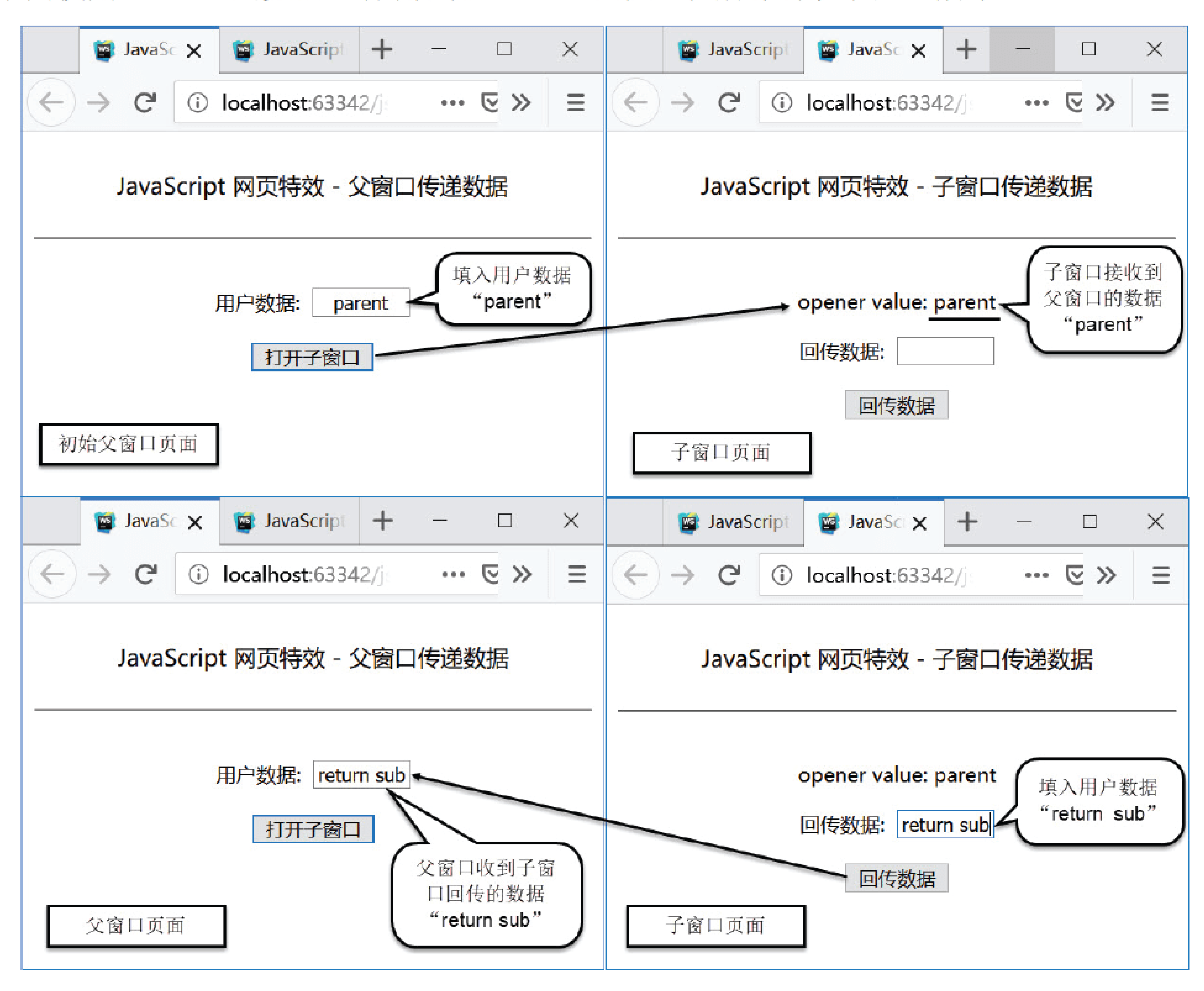


图 9.5 JavaScript 实现父子窗口之间数据交互

如图 9.5 中的箭头和标识所示，父子窗口之间通过 opener 属性成功实现了数据交互的操作。

9.6 刷新当前页面

对于经常上网的用户来讲，刷新浏览器页面是一项必不可少的操作。尤其是在页面变慢、迟迟无法响应的情况下，就会按下键盘上的“F5”键进行页面刷新。那么，通过 JavaScript 脚本语言如何实现页面的刷新操作呢？在 HTML DOM 中，Location 对象中定义有一个 reload()方法，用于重新加载当前文档。在本节中，就介绍如何通过 JavaScript 实现刷新当前页面的代码实例。

【代码 9-7】（详见源代码目录 ch09-js-html-refresh.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 网页特效 - 刷新当前页面</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
    <input type="button" value="人工方式刷新页面" onclick="refreshManual()"/>
    <input type="button" value="自动刷新页面" onclick="refreshAuto()"/>
</div>
</body>
<script type="text/javascript">
    function refreshManual() {
        var log = "manual refresh log : ";
        window.location.reload();
        var mdate = new Date();
        log += mdate.toLocaleString();
        console.log(log);
    }
    // TODO: define timer variables
    var timer = null;
    function refreshAuto() {
        var log = "init log : ";
        var pdate = new Date();
```

```
31          log += pdate.toLocaleString() + "\n";
32          console.log(log);
33          timer = self.setTimeout(function () {   // TODO: setTimeout
34              window.location.reload();
35              var adate = new Date();
36              log += "auto refresh log : " + adate.toLocaleString() + "\n";
37              console.log(log);
38              clearTimeout(timer);
39          }, 3000);
40      }
41  </script>
42  </html>
```

关于【代码 9-7】的说明：

- 第 14 行和第 15 行代码通过<input type="button">标签元素定义了两个按钮，分别用于实现“人工方式刷新页面”和模拟“自动刷新页面”的两种操作方式。
- 第 19 ~ 25 行代码是 refreshManual()方法的实现过程，用于实现“人工方式刷新页面”的操作。在第 21 行代码中，通过调用 Location 对象的 reload()方法来实现重新加载文档（等于刷新当前页面）。
- 第 28 ~ 40 行代码是 refreshAuto()方法的实现过程，用于实现模拟“自动刷新页面”的操作。其中，第 33 ~ 39 行代码通过使用 setTimeout()方法设定了一个时间延迟（3s），然后通过调用 Location 对象的 reload()方法来实现刷新当前页面。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 9.6 和图 9.7 所示。

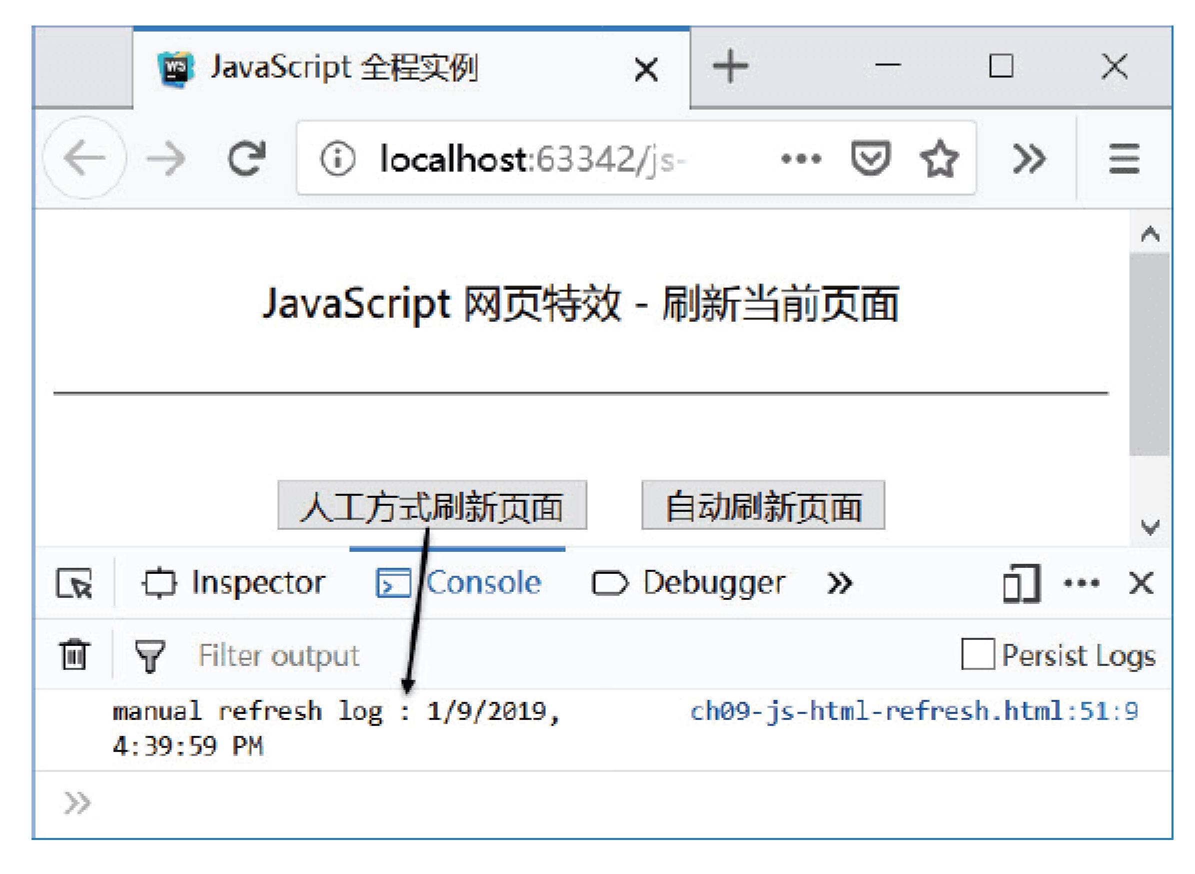


图 9.6　JavaScript 实现刷新当前页面（人工方式）

如图 9.6 中箭头所示，在单击“人工方式刷新页面”按钮后，浏览器控制台中显示了相应的信息。

如图 9.7 中箭头的标识所示，在单击“自动刷新页面”按钮后，浏览器控制台中分别显示了自动刷新前和自动刷新后的两行信息。

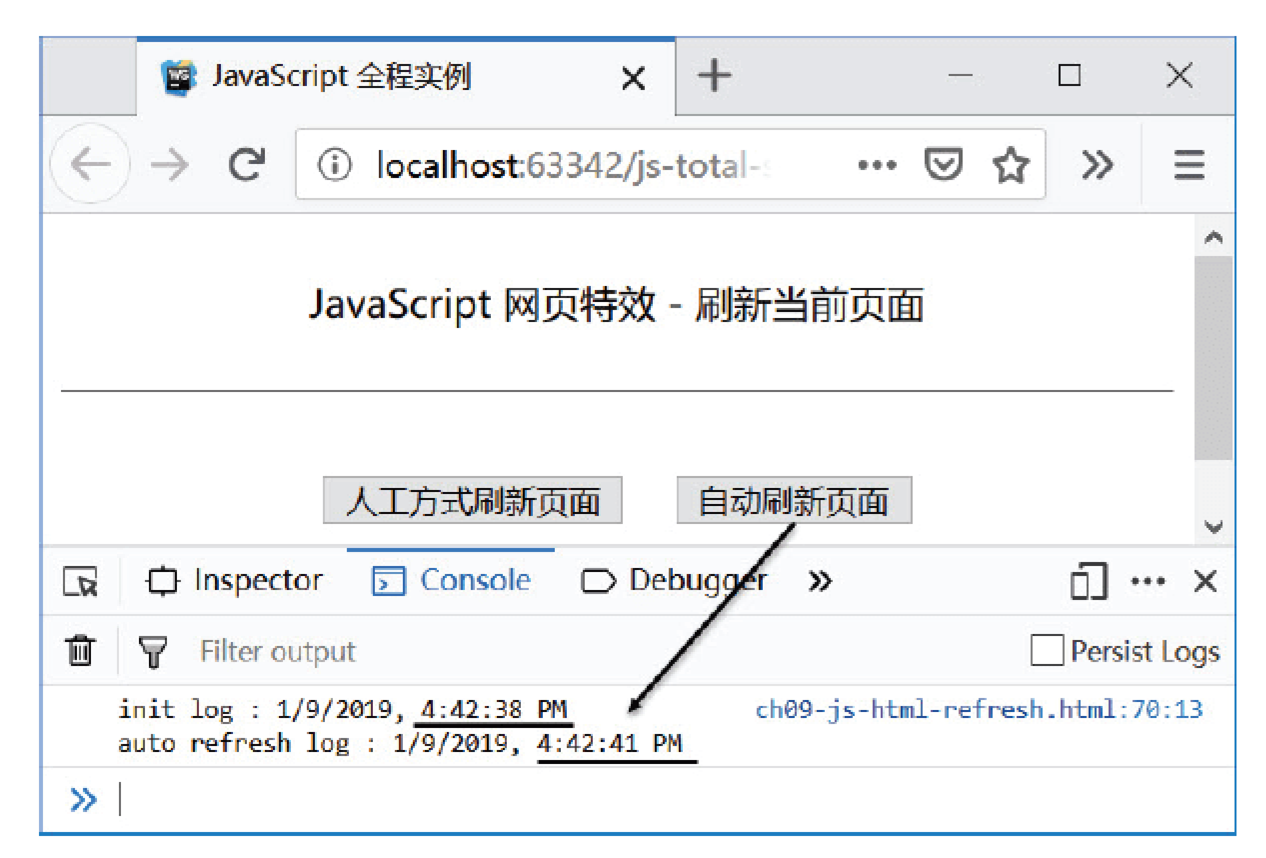


图 9.7 JavaScript 实现刷新当前页面（模拟自动）

9.7 屏蔽鼠标右键

在某些需要保护网页内容的网站中，通常是屏蔽掉鼠标右键的，也就是由用户尝试点击鼠标右键的操作是无效的。在本节中，介绍一个如何通过 JavaScript 实现屏蔽鼠标右键的代码实例。

【代码 9-8】（详见源代码目录 ch09-js-html-forbidden-rightmouse.html 文件）

```
01 <script type="text/javascript">
02     document.onmousedown = function (e) {
03         // TODO: 判断事件的值是否为鼠标右键
04         if (e.button == 2) {
05             alert('禁用鼠标右键!');   // TODO: 提示用户禁用鼠标右键
06         }
07     }
08 </script>
```

关于【代码 9-8】的说明：

- 第 02 ~ 07 行代码定义了监听 Document 对象中鼠标点击 onmousedown 事件的方法函数。其中，第 04 ~ 06 行代码通过判断 Event 对象的 button 属性值是否为 2（指代鼠标右键）来实现屏蔽鼠标右键的操作。

9.8 屏蔽上下文菜单

除了屏蔽鼠标右键操作，还有一个屏蔽上下文菜单的操作，二者在效果上是差不多的。对于屏蔽上下文菜单的操作，我们可以通过监听 oncontextmenu 事件来实现。在本节中，将介绍一个如何通过 JavaScript 实现屏蔽上下文菜单的代码实例。

【代码 9-9】（详见源代码目录 ch09-js-html-forbidden-contextmenu.html 文件）

```
01 <script type="text/javascript">
02     document.oncontextmenu = function (e) {
03         e.returnValue = false;
04     };
05 </script>
```

关于【代码 9-9】的说明：

- 第 02～04 行代码定义了监听 Document 对象中上下文菜单 oncontextmenu 事件的方法函数。其中，第 03 行代码通过设定 Event 对象的 returnValue 属性值为否（false）来实现屏蔽上下文菜单的操作。

9.9 屏蔽复制功能

本节将实现如何屏蔽网页中的复制功能。对于屏蔽网页中的复制操作，可以通过监听 oncopy 事件来实现。在本节中，将介绍一个如何通过 JavaScript 实现屏蔽网页复制功能的代码实例。

【代码 9-10】（详见源代码目录 ch09-js-html-forbidden-copy.html 文件）

```
01 <!doctype html>
02 <html lang="en">
03 <head>
04     <!-- 添加文档头部内容 -->
05     <title>JavaScript 全程实例</title>
06 </head>
07 <body oncopy="alert('禁止复制网页内容!');return false;">
08 <!-- 添加文档主体内容 -->
09 <header>
10     <nav>JavaScript 网页特效 - 禁止复制</nav>
11 </header>
12 </body>
13 </html>
```

关于【代码 9-10】的说明：

- 禁止网页复制的关键代码在第 07 行的<body>标签元素中，通过为 oncopy 事件方法定义返回值 false 来实现禁止复制内容的操作。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 9.8 所示。

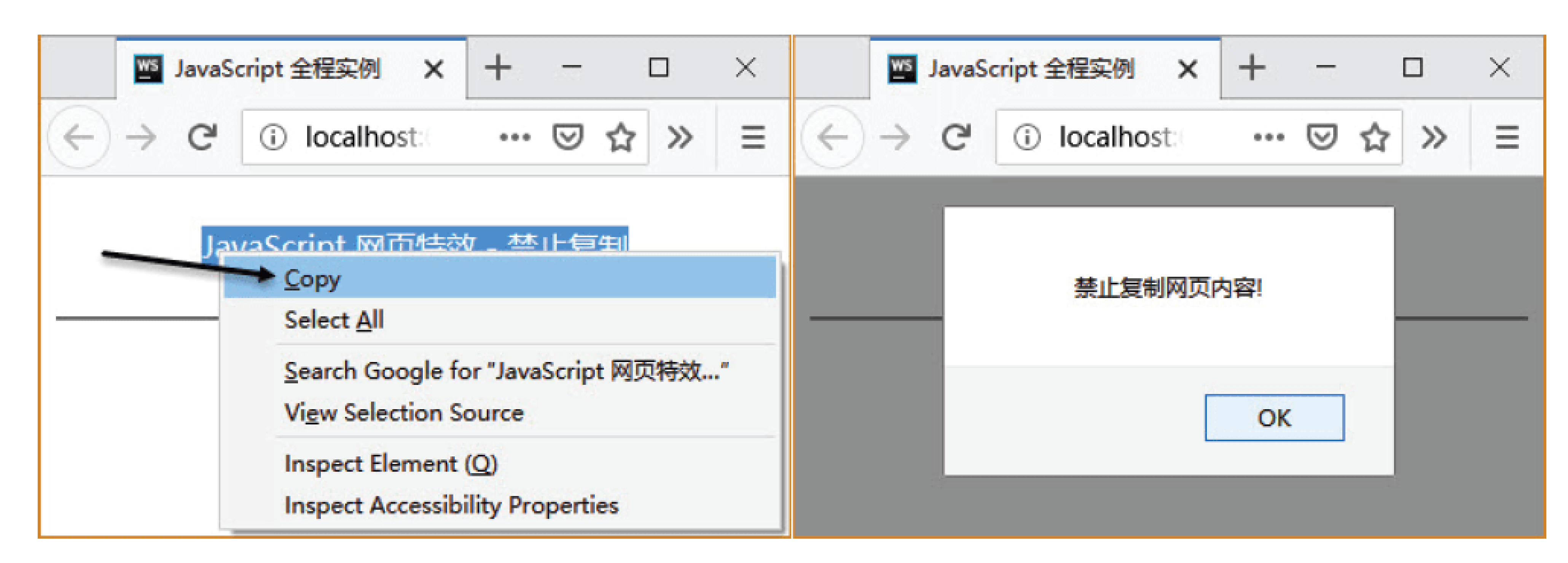


图 9.8　JavaScript 实现屏蔽复制功能

如图 9.8 中箭头所示，在选择页面内容并右击后，选择复制（copy）命令后，页面中弹出了警告提示框“禁止复制网页内容！”。那么，刚刚的复制操作到底成功没有呢？我们可以通过“剪切板”来查看，具体效果如图 9.9 所示。

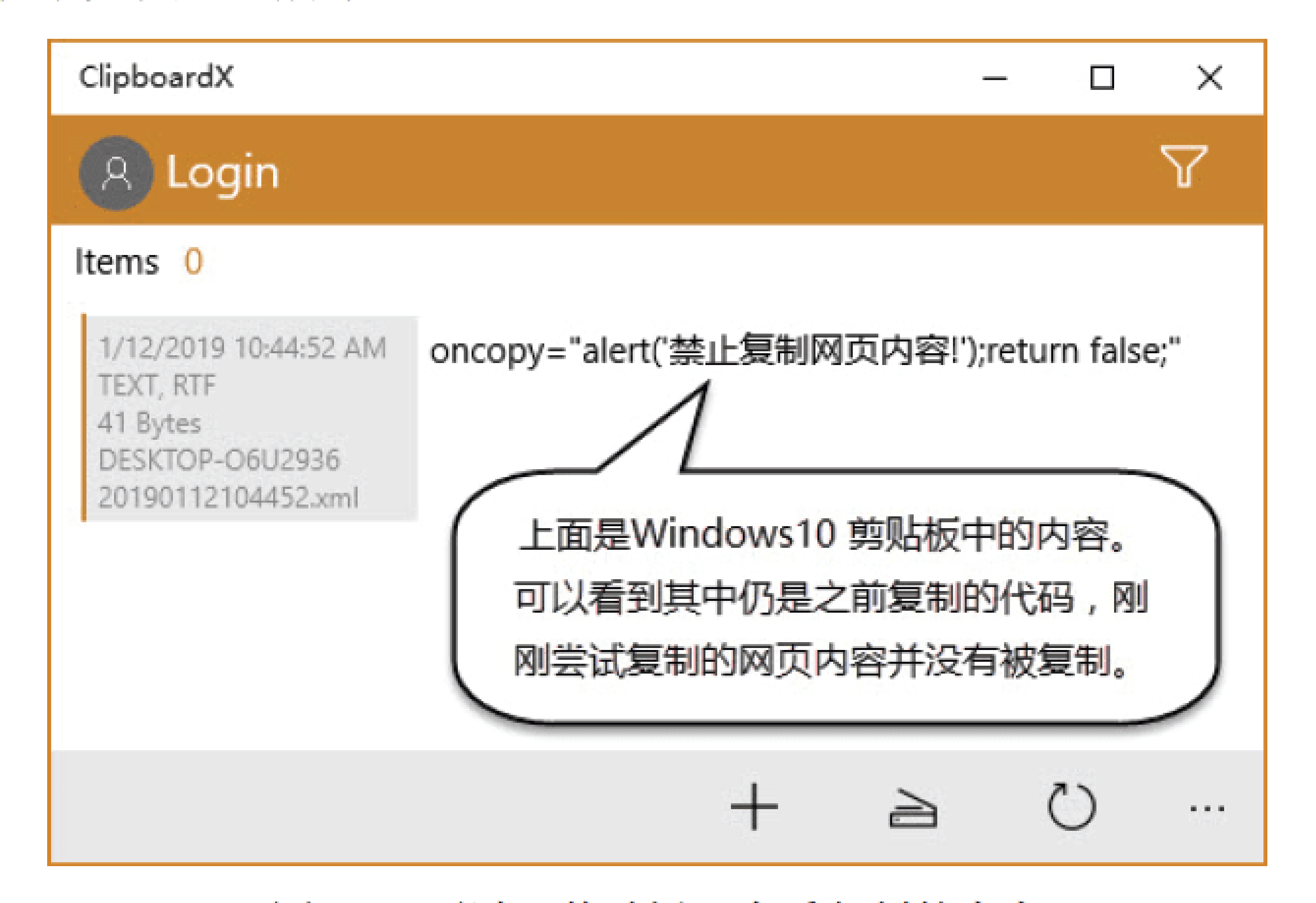


图 9.9　通过“剪贴板”查看复制的内容

如图 9.9 中标识所示，刚刚尝试复制的网页内容并没有成功，“剪切板”中的最近内容仍为之前复制的 JavaScript 代码。


提 示

某些 Windows 10 版本的系统中没有提供剪贴板应用（带可视化窗口），读者需要自行下载第三方的免费应用。

9.10　屏蔽选择操作

屏蔽网页中的复制功能，看起来似乎有点烦琐。其实，通过 JavaScript 脚本语言可以直接屏蔽网页中的选择操作（通过拖动鼠标选择）。无法选择，也就无法复制了。在网页中屏蔽选择操作，可以通过监听 onselectstart 事件来实现。在本节中，将介绍一个如何通过 JavaScript 实现屏蔽网页选择操作的代码实例。

【代码 9-11】（详见源代码目录 ch09-js-html-forbidden-select.html 文件）

```
01 <script type="text/javascript">
02     document.onselectstart = function (e) {
03         e.returnValue = false;
04     };
05 </script>
```

关于【代码 9-11】的说明：

- 第 02 ~ 04 行代码定义了监听 Document 对象中开始选择 onselectstart 事件的方法函数。
- 第 03 行代码通过设定 Event 对象的 returnValue 属性值为否（false）来实现屏蔽选择的操作。

如果在网页中屏蔽了选择操作，即使用户通过拖动鼠标进行选择，最终也是无法选取任何文本内容的。

9.11 防止网页被“frame”

总有一些无良的设计人员，喜欢直接嵌套（“frame”）他人的页面到自己的网站中。而且，这样做的成本是非常低的，直接通过<iframe>框架引用目标网址就可以实现。为了防止自己的页面被别人“frame”，可以在页面中嵌入一段 JavaScript 脚本，用于检测当前页面是不是顶层（top）窗口，假如检测出不是顶层（top）窗口，就通过脚本语言修改过来。这样，当别人“frame”你的网页时，就会自动转向你的网页了，而不是被对方“frame”到他们的网页中。

在本节中，将介绍一个如何通过 JavaScript 实现屏蔽网页选择操作的代码实例。

【代码 9-12】（详见源代码目录 ch09-js-html-forbidden-frame.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 网页特效 - 防止网页被“frame”</nav>
</header>
</body>
<script type="text/javascript">
    window.onload = function () {
        // TODO: 判断有没有 frame
```

```
16          if (top.location != self.location) {
17              alert("网页被‘frame’了!");
18              top.location = self.location;
19          } else {
20              console.log("网页未被‘frame’!");
21          }
22      };
23  </script>
24  </html>
```

关于【代码 9-12】的说明：

- 第 16～21 行代码通过 if 条件语句判断当前窗口对象（self）是否为顶层窗口（top）框架。如果判断结果为 true，就表示当前窗口未被“frame”；如果判断结果为 false，就表示当前窗口已经被“frame”了，此时可通过第 18 行代码将当前窗口跳出到顶层窗口。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 9.10 所示。

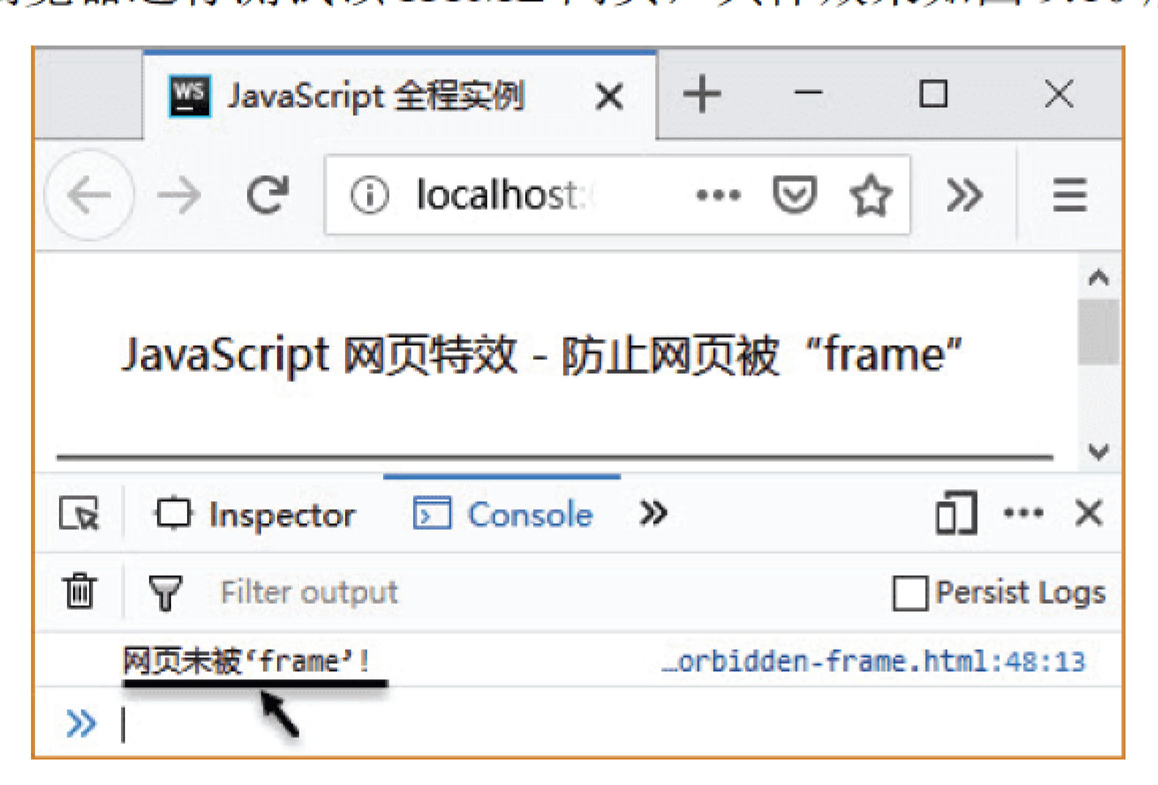


图 9.10　JavaScript 实现防止网页被“frame”（一）

如图 9.10 中箭头所示，浏览器控制台中提示“网页未被‘frame’”信息。网页被“frame”又是什么效果呢？下面看一个网页被“frame”后通过 JavaScript 跳出“frame”的代码实例。

【代码 9-13】（详见源代码目录 ch09-js-html-forbidden-frame-test.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 网页特效 - 防止网页被“frame”测试</nav>
```

```
11 </header>
12 <!-- 添加文档主体内容 -->
13 <div id="id-div-center" style="">
14     <iframe src="ch09-js-html-forbidden-frame.html"></iframe>
15 </div>
16 </body>
17 </html>
```

关于【代码 9-13】的说明：

- 第 14 行代码通过<iframe src="ch09-js-html-forbidden-frame.html">标签元素将【代码 9-12】定义的 HTML 页面"frame"进来了。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 9.11 所示。

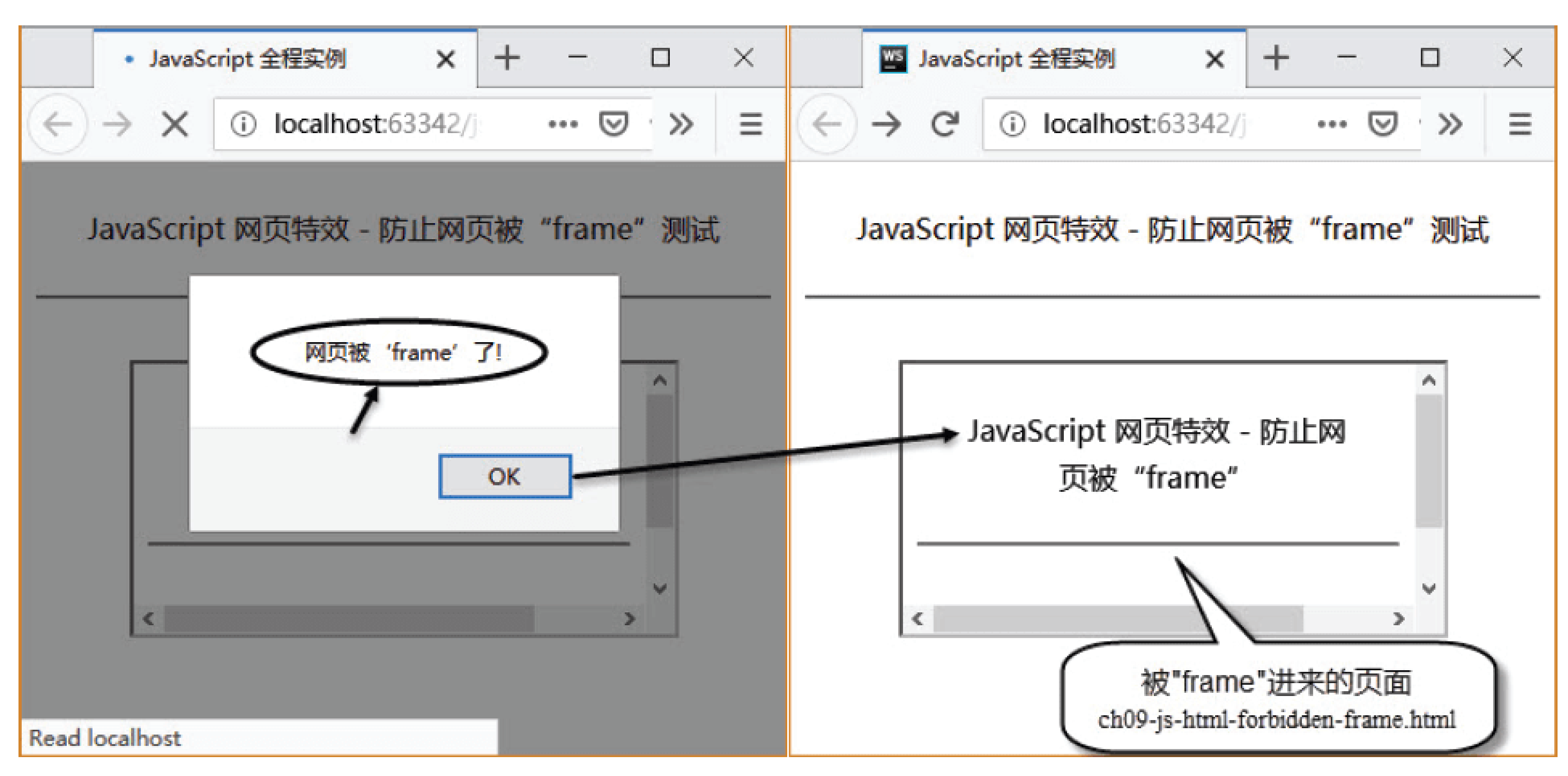


图 9.11　JavaScript 实现防止网页被"frame"（二）

如图 9.11 中箭头和标识所示，左边页面中显示了"网页被'frame'了！"的警告弹出框。在将警告弹出框关闭后，右边页面显示网页被"frame"了的效果。我们知道，【代码 9-12】中定义的第 18 行代码防止了页面（ch09-js-html-forbidden-frame.html）被"frame"。因此。图 9.11 中右边的页面很快会跳出并变成被"frame"的页面，如图 9.12 所示。

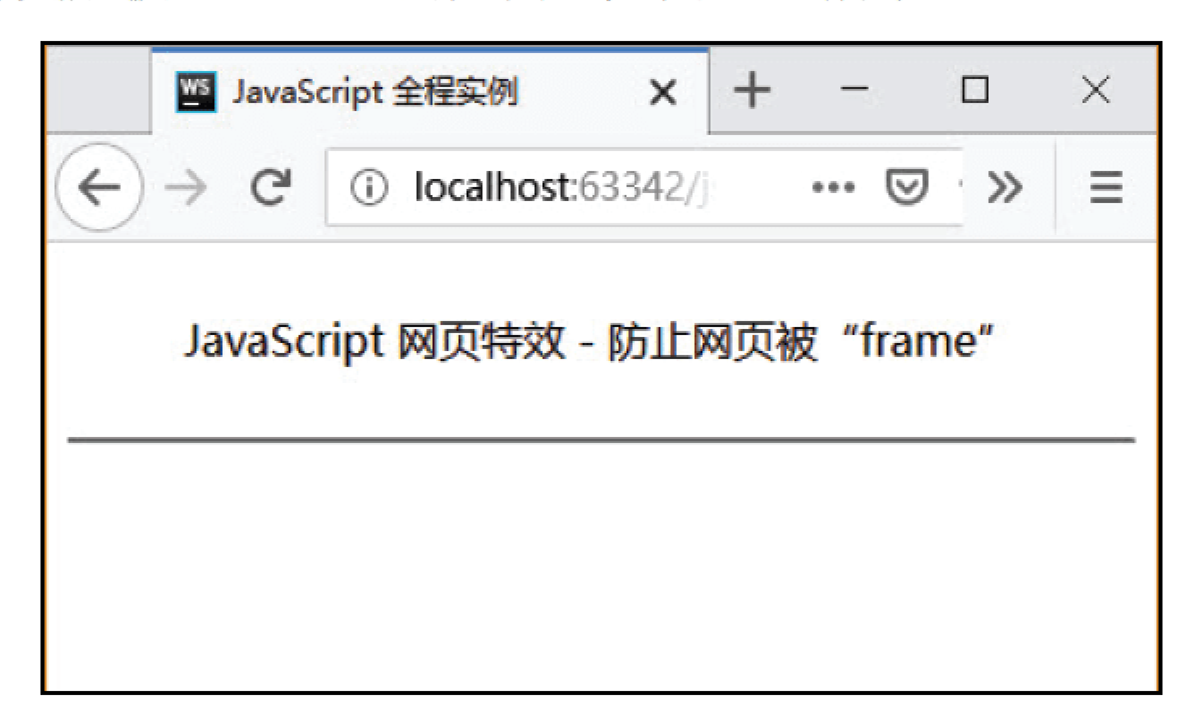


图 9.12　JavaScript 实现防止网页被"frame"（三）

9.12 隐藏页面滚动条

在默认情况下，当页面中的内容超过窗口边界时，会自动显示出滚动条（横向和竖向）的效果。但是，一般设计人员是不希望页面出现滚动条的（尤其是横向滚动条），这时就需要将滚动条隐藏掉。CSS 层叠样式表中提供了一个 overflow 属性，用于设置当内容溢出元素边界时如何处理。在本节中，将介绍一个如何通过 JavaScript 操作 overflow 属性来实现隐藏页面滚动条的代码实例。

【代码 9-14】（详见源代码目录 ch09-js-html-hidden-scrollbar.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 网页特效 - 隐藏页面滚动条</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
    <span>隐藏页面滚动条</span>
<input type="radio" id="id-scrollbar-1" value="1"
                              onclick="sbCheck(this.id)">允许页面滚动条
<input type="radio" id="id-scrollbar-0" value="0"
                              onclick="sbCheck(this.id)">隐藏页面滚动条
</div>
</body>
<script type="text/javascript">
    window.onload = function(){
        document.getElementById('id-scrollbar-1').checked = true;
    };
    function on_scrollbar_check(thisid) {
        var c = document.getElementById(thisid).value;
        switch(c) {
            case "0":
                // TODO: 隐藏且禁用横向纵向两个滚动条
                document.body.style.overflow = "hidden";
                break;
            case "1":
                // TODO: 开启横向纵向两个滚动条
```

```
32                  document.body.style.overflow = "auto";
33                  break;
34          }
35      }
36  </script>
37  </html>
```

关于【代码 9-14】的说明：

- 第 15 ~ 16 行代码通过<input type="radio">标签元素定义了一组（两个）单选按钮，并添加了单击 onclick 事件方法（sbCheck()），分别用于执行选择显示或隐藏页面滚动条的操作。
- 第 23 ~ 35 行代码是单击 onclick 事件方法（sbCheck()）的实现过程。其中，第 28 行和第 32 行代码分别根据用户的选择，通过设定 overflow 属性值实现显示（"auto"）或隐藏（"hidden"）滚动条。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 9.13 所示。

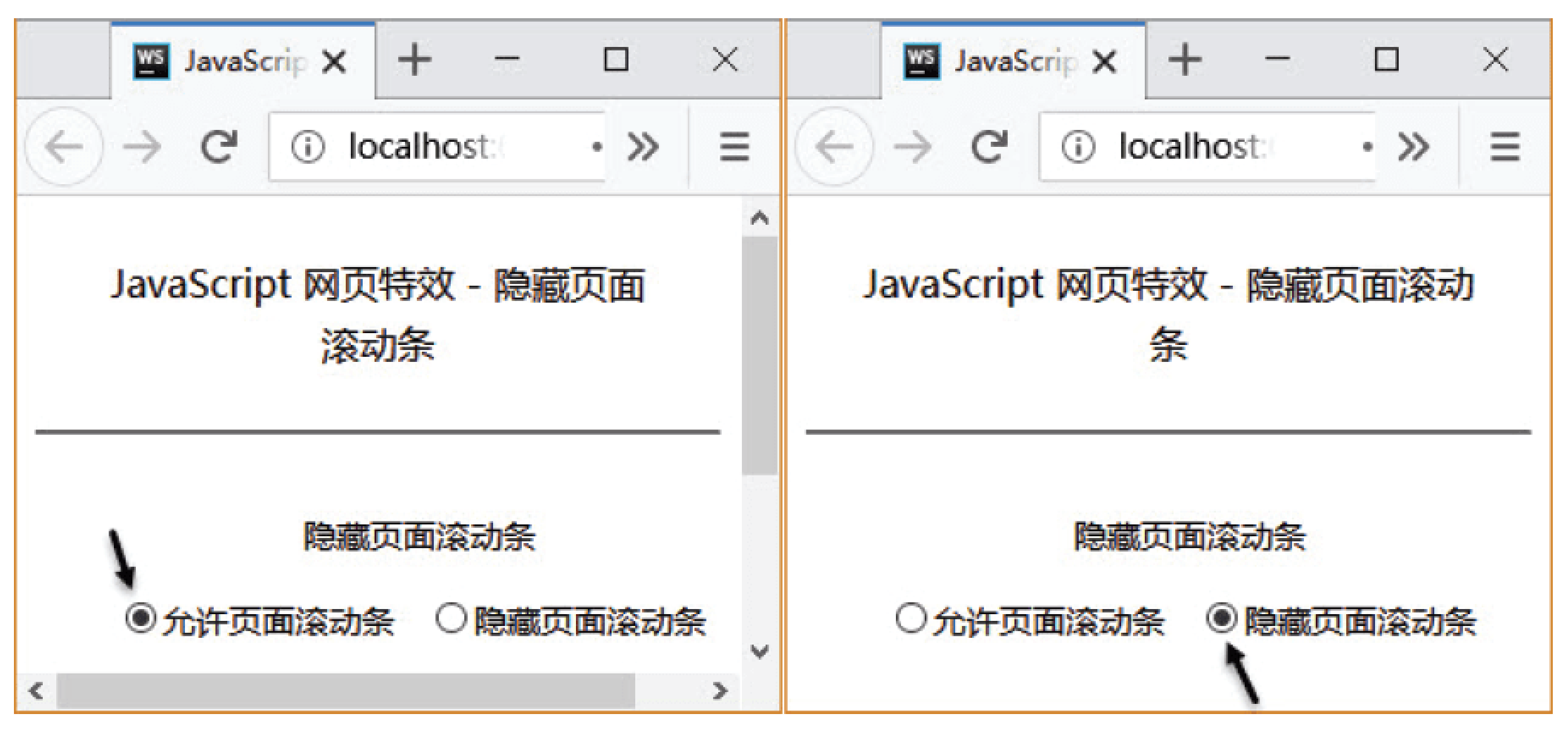


图 9.13　JavaScript 实现隐藏页面滚动条

如图 9.13 中箭头所示，左边页面中是默认显示滚动条的效果，右边页面中是隐藏滚动条的效果。另外，用户在左边页面是可以通过鼠标操作滚动条的，而在右边页面是无法进行滚动的。

9.13　最小化、最大化和关闭窗口

在某些情况下，需要将新打开的窗口以最小化或最大化的方式显示出来。HTML DOM 中提供了一个 resizeTo()方法，可以按照指定的窗口尺寸进行调整。另外，如果需要关闭窗口，可以通过 close()方法来实现。

下面介绍一个如何通过 JavaScript 实现以最小化和最大化方式打开窗口以及关闭窗口操作的代码实例。

【代码 9-15】（详见源代码目录 ch09-js-html-min-max-close.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 网页特效 - 最小化、最大化和关闭窗口</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
    <input type="button" value="最小化窗口" onclick="window_min()"/><br><br>
    <input type="button" value="最大化窗口" onclick="window_max()"/><br><br>
    <input type="button" value="关闭窗口"onclick="window_close()"/><br><br>
</div>
</body>
<script type="text/javascript">
    var win;   // TODO: define global window obj
    function window_min() {
        win = window.open("open.html", "", "_blank","width=350,
                                 height=300");
        win.focus();
        var w = 1, h = 1;
        win.resizeTo(w, h);
    }
    function window_max() {
        win = window.open("open.html", "", "_blank", "width=350,
                                 height=300");
        win.focus();
        var w = 0, h = 0;
        w = window.screen.availWidth;
        h = window.screen.availHeight;
        win.moveTo(0, 0);
        win.resizeTo(w, h);
    }
    function window_close() {
        win.focus();
        win.close();
    }
```

```
40 </script>
41 </html>
```

关于【代码 9-15】的说明：

- 第 20 行代码定义了一个全局变量（win），用于保存通过 window.open()方法打开的窗口对象。
- 第 21 ~ 26 行代码定义的单击 onclick 事件方法（window_min()）用于实现以最小化的方式打开窗口的操作。其中，第 22 行代码通过 window.open()方法打开本地 open.html 页面，并定义了窗口的打开方式（"_blank"）和初始化尺寸（"width=350,height=300"）；第 24 行代码定义了最小化窗口的尺寸（w=1, h=1）；第 25 行代码通过调用 resizeTo()方法以最小化方式重新调整新打开的 open.html 页面。
- 第 27 ~ 35 行代码定义的单击 onclick 事件方法（window_max()）用于实现以最大化的方式打开窗口的操作。其中，第 28 行代码通过 window.open()方法打开本地 open.html 页面（同第 22 行代码）；第 31 行和第 32 行代码通过 Screen 对象的 availWidth 属性和 availHeight 属性获取了设备屏幕的可用尺寸；第 33 行代码通过调用 moveTo()方法将 open.html 页面窗口移动到屏幕原点（0，0）；第 34 行代码通过调用 resizeTo()方法以最大化（屏幕可用尺寸）方式重新调整该窗口尺寸。
- 第 36 ~ 39 行代码定义的单击 onclick 事件方法（window_close()）用于实现关闭窗口的操作，主要是通过第 38 行代码调用的 close()方法来实现的。

9.14 脚本永不出错

如果加入 HTML 页面中的 JavaScript 脚本存在错误，那么在页面打开运行过程中可能会出现用户并不希望看到的提示脚本错误的弹出对话框，这是一种不太友好的 JavaScript 语言设计方式。为避免这种情况，可以在网页中通过使用 onerror 事件方法来添加捕获错误的代码，理想情况下是可以让 JavaScript 脚本永不出错。下面，我们就介绍一个如何通过 JavaScript 实现脚本永不出错的代码实例。

【代码 9-16】（详见源代码目录 ch09-js-html-onerror.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 网页特效 - 脚本永不出错</nav>
```

```
11 </header>
12 <!-- 添加文档主体内容 -->
13 <div id="id-div-center" style="">
14     <input type="button" id="id-input-error" value="测试错误捕获"
               onclick="testErr();"/>
15 </div>
16 </body>
17 <script type="text/javascript">
18     window.onload = function (ev) {
19         window.onerror = function (msg, url, line) {
20             console.log(msg);
21             console.log(url);
22             console.log(line);
23             // TODO: 屏蔽脚本错误提示
24             return true;
25         }
26     };
27     function testErr() {
28         aaalert("测试错误捕获");
29     }
30 </script>
31 </html>
```

关于【代码 9-16】的说明：

- 第 14 行代码通过<input>标签元素定义了一个按钮，并定义了单击 onclick 事件处理方法（testErr()）。
- 第 27 ~ 29 行代码是单击 onclick 事件方法（testErr()）的实现过程。其中，第 28 行代码定义了弹出警告框，但是 alert()方法名写错了（写成 aaalert()了），目的就是人为制造出一个脚本错误，用于测试错误捕获。
- 关键代码是第 19 ~ 25 行定义的错误捕获 onerror 事件方法，该方法中的三个参数分别表示错误信息（msg）、错误地址（url）和错误代码行数（line）。第 24 行代码定义的返回值（true）是屏蔽脚本错误提示的关键。下面我们将分别测试一下有无定义该行代码的不同结果。

使用 Firefox 浏览器运行测试该 HTML 网页（这里先注释掉第 24 行代码），具体效果如图 9.14 所示。

如图 9.14 中箭头所示，浏览器控制台中分别显示了错误信息、错误地址和错误代码行数，并以显著的方式给出了错误提示。

下面继续使用 Firefox 浏览器运行测试该 HTML 网页（这里恢复第 24 行代码），具体效果如图 9.15 所示。

如图 9.15 中所示，浏览器控制台中分别显示了错误信息、错误地址和错误代码行数，但并没有给出如图 9.14 中的错误提示，说明脚本错误提示被屏蔽掉了。

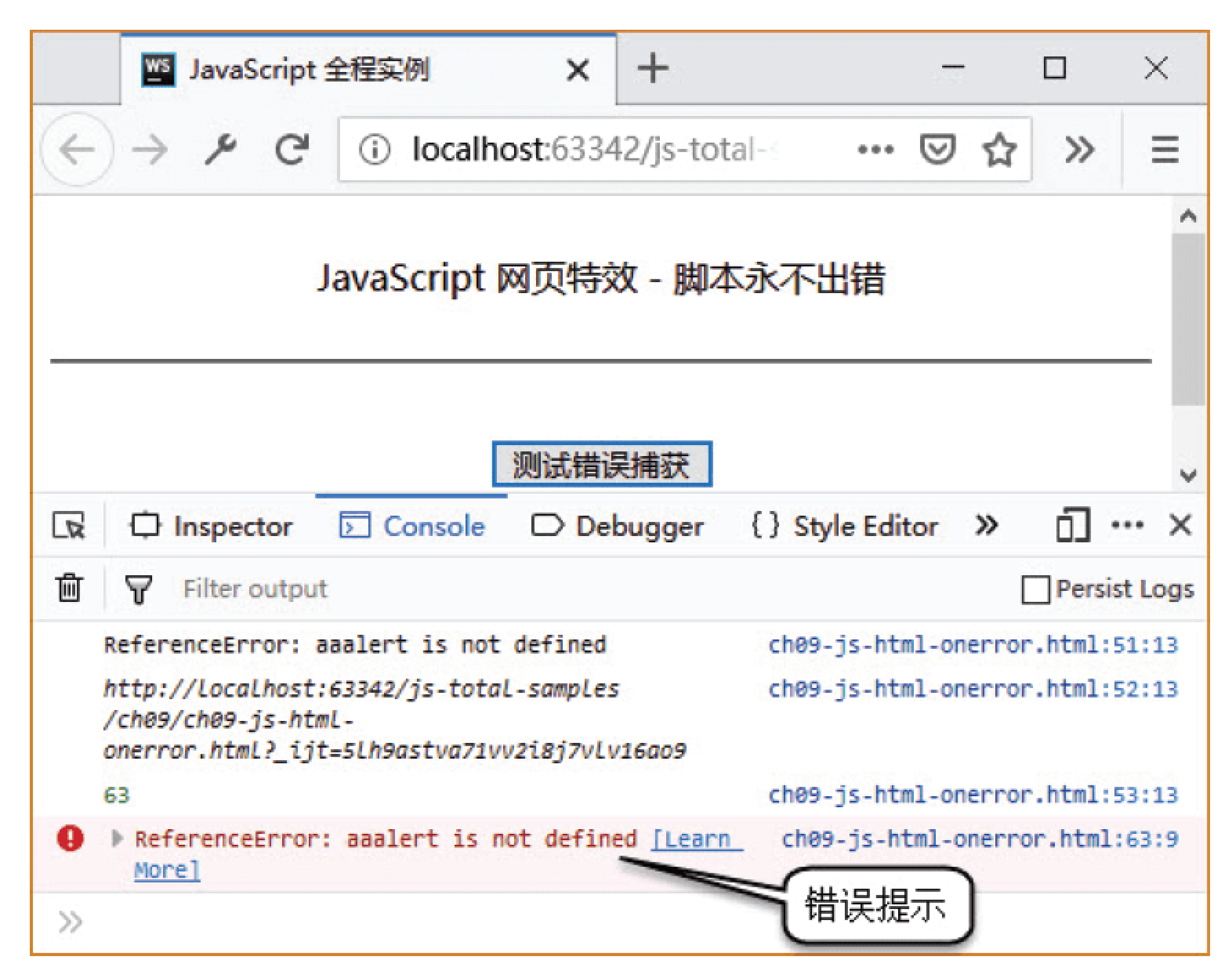


图 9.14　JavaScript 实现脚本永不出错（一）

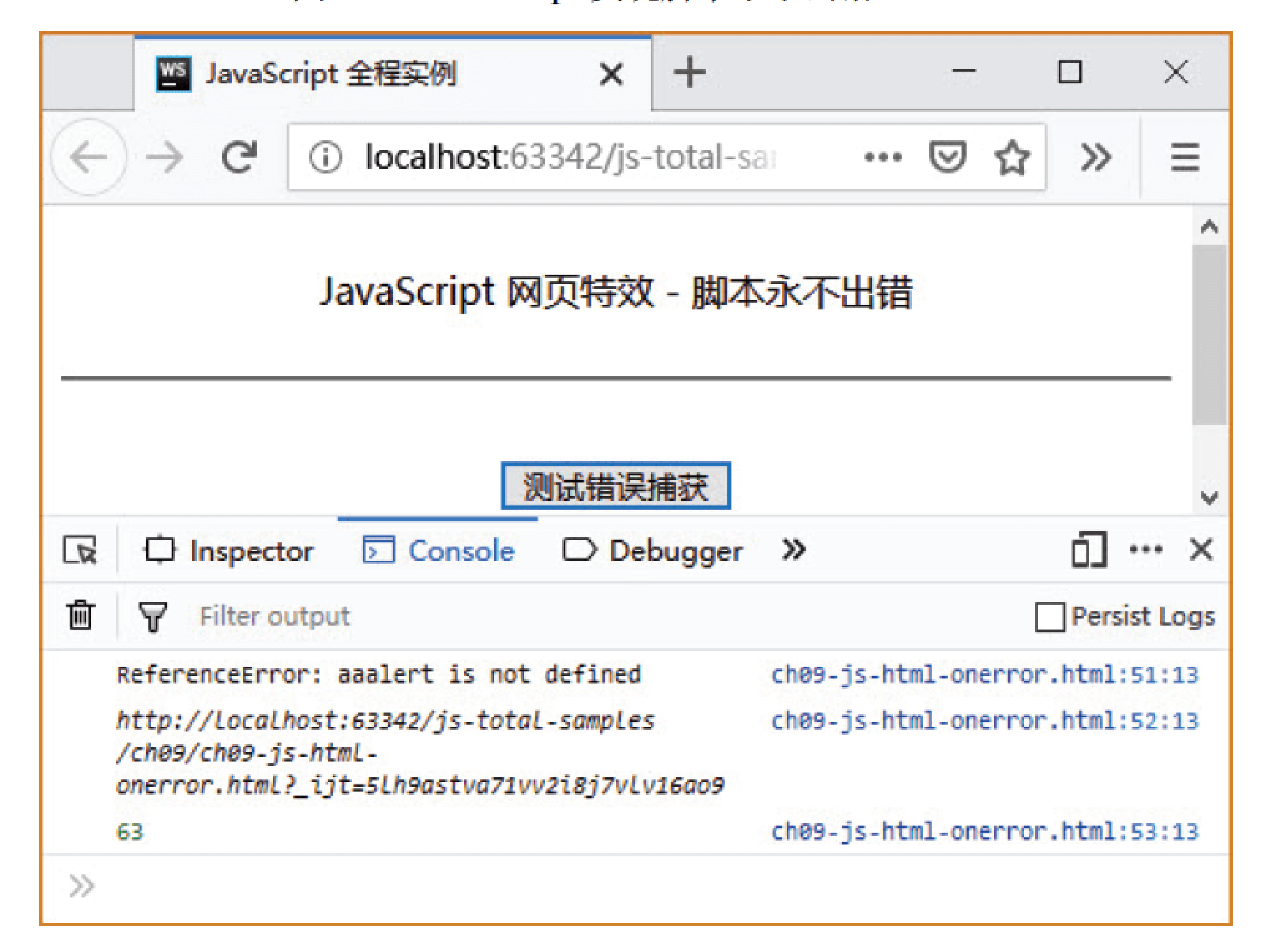


图 9.15　JavaScript 实现脚本永不出错（二）

9.15　获取浏览器信息

各大主流浏览器的开发商在设计浏览器时均会提供关于浏览器内核、类型、代码名称、版本号、语言等方面的信息。如果想获取这些信息，通过使用 Navigator 对象就可以实现。下面，我们介绍一个如何通过 JavaScript 实现获取浏览器信息的代码实例。

【代码 9-17】（详见源代码目录 ch09-js-html-get-browser-info.html 文件）

```
01 <!doctype html>
02 <html lang="en">
03 <head>
04     <!-- 添加文档头部内容 -->
05     <title>JavaScript 全程实例</title>
06 </head>
07 <body>
08 <!-- 添加文档主体内容 -->
09 <header>
10     <nav>JavaScript 网页特效 - 获取浏览器信息</nav>
11 </header>
12 <!-- 添加文档主体内容 -->
13 <div id="id-div-center" style="">
14     <span id="id-span-browser-info"></span>
15 </div>
16 </body>
17 <script type="text/javascript">
18     window.onload = function () {
19         var browser_info = "<b>浏览器主要信息:</b>" + "<br>";
20         browser_info += "浏览器代码名称: " + navigator.appCodeName + "<br>";
21         browser_info += "浏览器类型: " + navigator.appName + "<br>";
22         browser_info += "浏览器内核: " + navigator.userAgent + "<br>";
23         browser_info += "浏览器版本: " + navigator.appVersion + "<br>";
24         browser_info += "浏览器语言: " + navigator.language + "<br><br>";
25         browser_info += "<b>浏览器全部信息:</b>" + "<br>";
26         for (var n in navigator) {
27             if (typeof navigator[n] == "string")
28                 browser_info += n + " : " + navigator[n] + "<br>";
29         }
30         document.getElementById("id-span-browser-info").innerHTML =
                                                              browser_info;
31     };
32 </script>
33 </html>
```

关于【代码 9-17】的说明：

- 第 20 ~ 24 行代码通过 Navigator 对象的一组重要属性，分别获取浏览器代码名称（appCodeName）、浏览器类型（appName）、浏览器内核（userAgent）、浏览器版本（appVersion）和浏览器语言（language）这几个主要信息。

- 第 26~29 行代码通过 for 循环迭代语句，遍历了 Navigator 对象中全部字符类型的属性，相当于获取了关于浏览器的全部重要信息。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 9.16 所示。

如图 9.16 中箭头所示，userAgent 属性中包含了关于浏览器内核的重要信息，因此设计人员在编写浏览器兼容性测试代码时主要就是依据 userAgent 属性。

这里需要读者特别注意的是，Navigator 对象是属于 HTML BOM（浏览器对象模型）范畴的。我们都知道 BOM 并不是正式的 W3C 标准，因此获取的浏览器信息均是浏览器开发商自己定义的，并不是标准化信息。

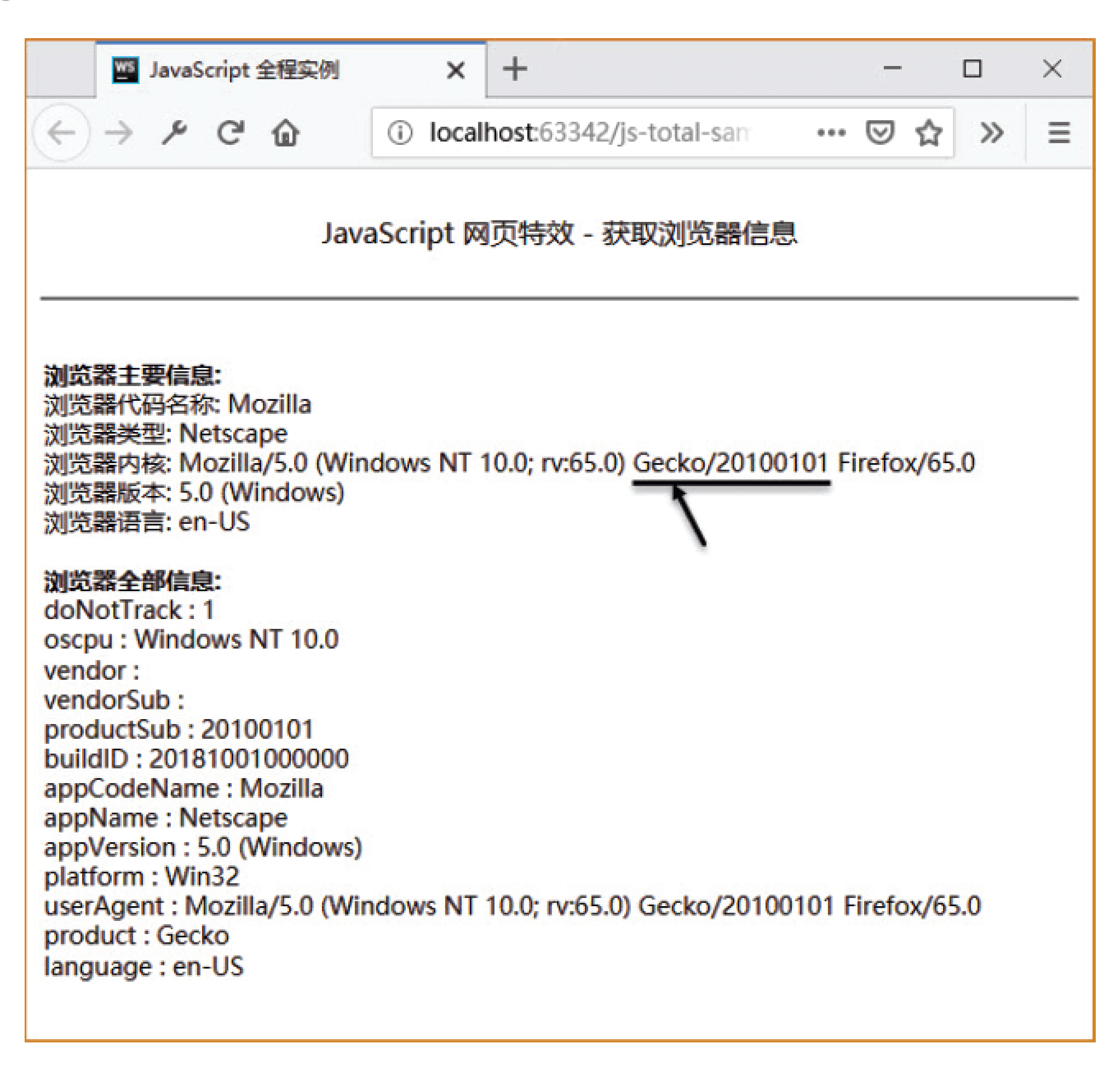


图 9.16　JavaScript 实现获取浏览器信息

9.16　获取浏览器窗口尺寸

本节我们介绍如何获取浏览器窗口的大小尺寸（宽度和高度），特别区分了浏览器窗口的整体外部尺寸和内部可视化尺寸。下面介绍一个如何通过 JavaScript 实现获取浏览器窗口尺寸的代码实例。

【代码 9-18】（详见源代码目录 ch09-js-html-get-window-size.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
```

```
06 </head>
07 <body>
08 <!-- 添加文档主体内容 -->
09 <header>
10     <nav>JavaScript 网页特效 - 获取浏览器窗口尺寸</nav>
11 </header>
12 <!-- 添加文档主体内容 -->
13 <div id="id-div-center" style="">
14     <span id="id-span-window-size"></span>
15 </div>
16 </body>
17 <script type="text/javascript">
18     window.onload = function () {
19         var s_info = "<b>浏览器窗口尺寸:</b>" + "<br><br>";
20         var w_outer_size = getWindowOuterSize();
21         s_info += "window.outerWidth: " + w_outer_size.width + " / " +
                   "window.outerHeight: " + w_outer_size.height + "<br>";
22         var w_inner_size = getWindowInnerSize();
23         s_info += "window.innerWidth: " + w_inner_size.width + " / " +
                   "window.innerHeight: " + w_inner_size.height + "<br><br>";
24         document.getElementById("id-span-window-size").innerHTML = s_info;
25     };
26     function getWindowOuterSize() {
27         return {
28             width: window.outerWidth,
29             height: window.outerHeight
30         };
31     }
32     function getWindowInnerSize() {
33         return {
34             width: window.innerWidth || document.documentElement.clientWidth,
35             height: window.innerHeight || document.documentElement.
                                                              clientHeight
36         };
37     }
38 </script>
39 </html>
```

关于【代码 9-18】的说明：

- 第 26 ~ 31 行代码定义的自定义方法 getWindowOuterSize()用于获取浏览器窗口整体外部尺寸，这里使用了 Window 对象的 outerWidth 属性和 outerHeight 属性。

- 第 32～37 行代码定义的自定义方法 getWindowInnerSize()用于获取浏览器窗口内部可视化尺寸，这里使用了 Window 对象的 innerWidth 属性和 innerHeight 属性。同时，还使用了 Document 对象的 clientWidth 属性和 clientHeight 属性，主要是用来保证早期 IE 浏览器版本的兼容性。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 9.17 所示。

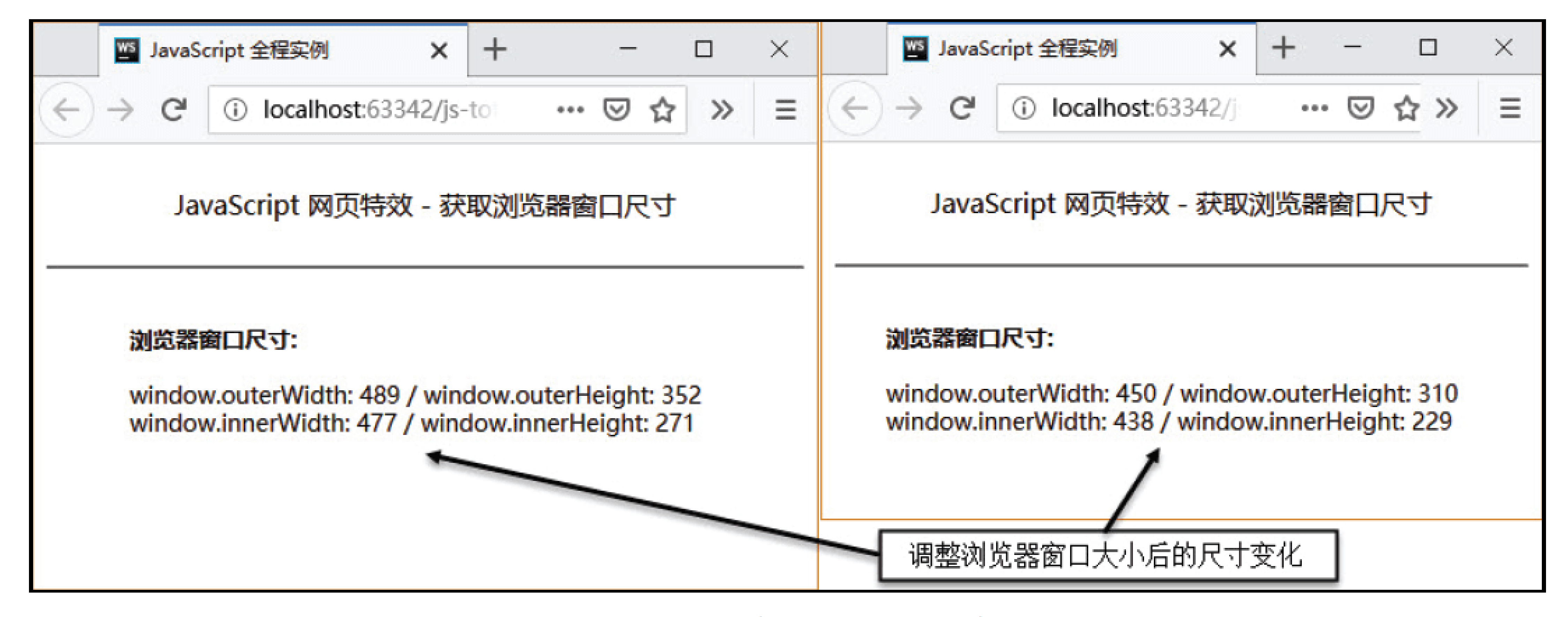


图 9.17　JavaScript 实现获取浏览器窗口尺寸

9.17　屏蔽键盘功能键

本节我们介绍如何在浏览器窗口中屏蔽功能键，主要测试 Shift、Ctrl 和 Alt 这三个功能键。下面看一下具体的 JavaScript 代码实例。

【代码 9-19】（详见源代码目录 ch09-js-html-forbidden-fkey.html 文件）

```
<script type="text/javascript">
    window.onload = function (ev) {
        document.onkeydown = function (ev) {
            var event = ev || window.event;
            if (event.shiftKey) {
                alert("禁止按 Shift 键!");
                return false;
            } else if (event.ctrlKey) {
                alert("禁止按 Ctrl 键!");
                return false;
            } else if (event.altKey) {
                alert("禁止按 Alt 键!");
                return false;
            } else {}
            console.log(event.key);
        };
```

```
17        };
18    </script>
```

关于【代码 9-19】的说明：

- 第 03 ~ 16 行代码通过监听 Document 对象的键盘按键 onkeydown 事件判断用户是否按下了 Shift（event.shiftKey）、Ctrl（event.ctrlKey）和 Alt（event.altKey）这三个功能键，然后通过 return 返回值（false）屏蔽这三个功能键。其中，第 15 行代码增加了一行控制台的输出信息，用于记录用户的按键操作（event.key）过程。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 9.18 所示。

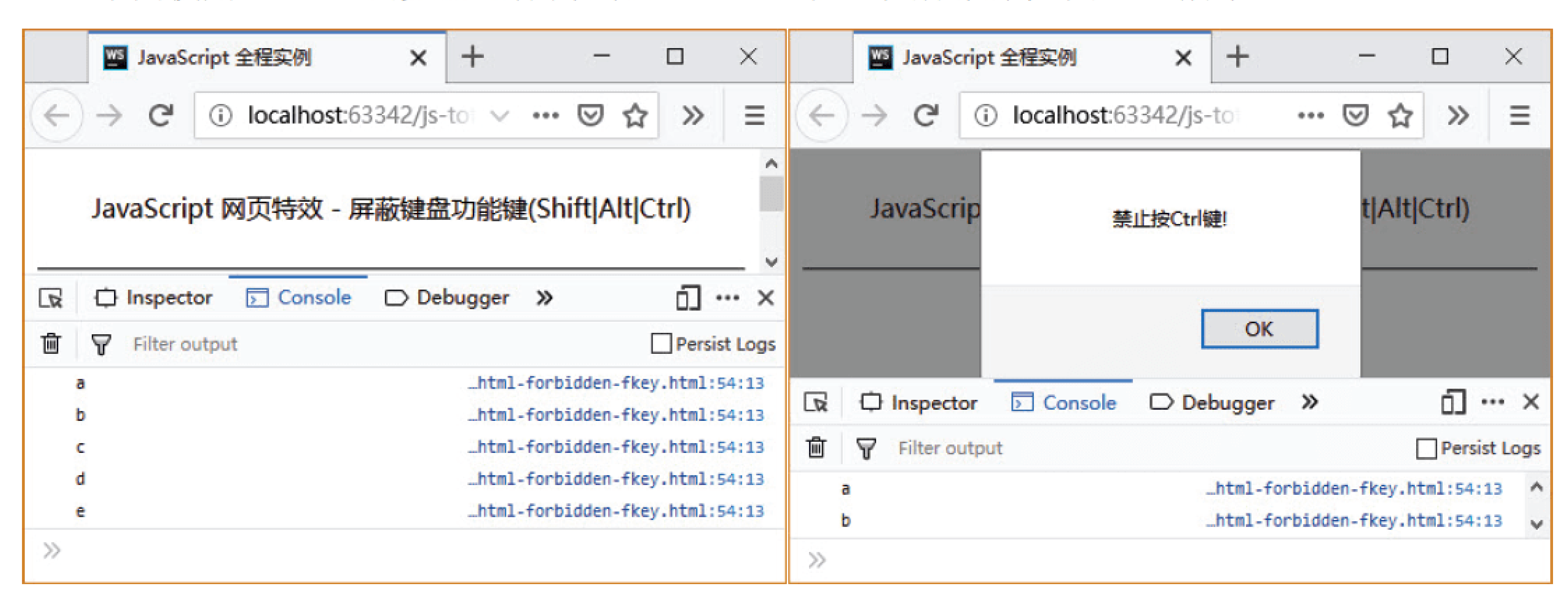


图 9.18　JavaScript 实现屏蔽键盘功能键

如图 9.18 中所示，在左边页面的浏览器控制台中显示了用户操作的一些按键信息，在右边页面中演示了当用户尝试按下 Ctrl + C 功能键进行复制操作后的效果（Ctrl 键被屏蔽了）。

9.18　页面窗口动画缩放

本节我们将实现一个让页面窗口动画缩放的功能，主要是通过 Window 对象的 resizeBy()方法按照设定的时间间隔动态调整窗口尺寸，达到页面窗口慢慢放大和缩小的效果。下面看一下具体的 JavaScript 代码实例。

【代码 9-20】（详见源代码目录 ch09-js-html-window-ani-size.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
```

```
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 网页特效 - 页面窗口动画缩放</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
    设定缩放尺寸:  
    <input type="text" id="id-input-size" value=""/>
    <input type="button" value="动画放大窗口"
            onclick="ani_large('id-input-size');"/>
    <input type="button" value="动画缩小窗口"
            onclick="ani_small('id-input-size');"/>
</div>
</body>
<script type="text/javascript">
    var timerLarge = null, timerSmall = null;
    var oWidth = 0, oHeight = 0;
    var scrWidth = screen.availWidth;
    var srcHeight = screen.availHeight;
    function ani_large(thisid) {
        var size = document.getElementById(thisid).value;
        var win = window.open("open.html", "", "_blank", "width=350,
                                height=300");
        win.moveTo(0, 0);
        timerLarge = setInterval(function () {
            win.resizeBy(size, size);
            oWidth = win.outerWidth;
            oHeight = win.outerHeight;
            win.location.reload();
            if (oWidth >= scrWidth || oHeight >= srcHeight) {
                clearInterval(timerLarge);
                timerLarge = null;
            }
        }, 1000);
    }
    function ani_small(thisid) {
        var size = document.getElementById(thisid).value;
        var win = window.open("open.html", "", "_blank", "width=600,
                                height=450");
        win.moveTo(0, 0);
        timerSmall = setInterval(function () {
            win.resizeBy(-size, -size);
            oWidth = win.outerWidth;
            oHeight = win.outerHeight;
```

```
48             win.location = win.location;
49             if (oWidth < 10 || oHeight < 10) {
50                 clearInterval(timerSmall);
51                 timerSmall = null;
52             }
53         }, 1000);
54     }
55 </script>
56 </html>
```

关于【代码 9-20】的说明：

- 第 15 行代码通过<input>标签元素定义了一个文本框，用于设置窗口动画缩放的调整尺寸。
- 第 16 行和第 17 行代码分别通过<input>标签元素定义了两个按钮，一个用于动画放大窗口（ani_large()），一个用于动画缩小窗口（ani_small()）。
- 第 25～39 行代码是 ani_large()事件方法的实现过程。其中，第 29～38 行代码定义了一个计时器。第 30 行代码通过调用 Window 对象的 resizeBy()方法调整页面窗口尺寸大小。另外，第 34～37 行代码通过 if 条件语句判断调整后的窗口尺寸是否超出了屏幕可用尺寸，如果超出了就通过清除计时器停止窗口放大操作。
- 第 40～54 行代码是 ani_small()事件方法的实现过程，原理与 ani_large()事件方法类似，在调整后的窗口尺寸小于数值 10 后停止窗口缩小操作。

9.19 定时关闭页面

本节我们实现一个定时关闭页面窗口的功能，具体方法是先设定一个定时器，再通过调用 Window 对象的 close()方法来实现关闭页面窗口的操作。下面看一下具体的 JavaScript 代码实例。

【代码 9-21】（详见源代码目录 ch09-js-html-time-close-window.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript 网页特效 - 定时关闭页面</nav>
</header>
<!-- 添加文档主体内容 -->
```

```
13 <div id="id-div-center" style="">
14     设定定时关闭窗口时间：
15     <input type="text" id="id-input-time" value=""/>
16     <input type="button" value="定时关闭页面" onclick="set_time_close
                                                ('id-input-time');"/>
17 </div>
18 </body>
19 <script type="text/javascript">
20     var timer = null;
21     var win = null;
22     function set_time_close(id) {
23         var t = document.getElementById(id).value;
24         var timeClose = t || -1;
25         if (timeClose != -1) {
26             timer = setTimeout(function () {
27                 win = window.open('', '_self', '');
28                 win.close();
29                 timer.clearTimeout();
30             }, timeClose);
31         }
32     }
33 </script>
34 </html>
```

关于【代码 9-21】的说明：

- 第 15 行代码通过<input>标签元素定义了一个文本框，用于设置定时关闭页面的延迟时长。
- 第 16 行代码通过<input>标签元素定义了一个按钮，并定义了单击 onclick 事件处理方法（set_time_close()），用于执行定时关闭页面的操作。
- 第 22 ~ 32 行代码是 set_time_close()事件方法的实现过程。其中，第 26 ~ 30 行代码定义了一个计时器，先在第 27 行代码中通过调用 Window 对象的 open()方法重新打开一次当前页面，再在第 28 行代码中通过调用 Window 对象的 close()方法关闭页面。当然关闭页面操作是在定时器执行完毕后才完成的。

9.20 修改浏览器标题

本节我们介绍一下如何修改浏览器的标题（浏览器标题是通过<title>标签元素定义的），通过修改 Document 对象的 title 属性就可以完成修改浏览器标题。下面看一下具体的 JavaScript 代码实例。

【代码 9-22】（详见源代码目录 ch09-js-html-time-close-window.html 文件）

```
01 <!doctype html>
02 <html lang="en">
03 <head>
04     <!-- 添加文档头部内容 -->
05     <title>JavaScript 全程实例</title>
06 </head>
07 <body>
08 <!-- 添加文档主体内容 -->
09 <header>
10     <nav>JavaScript 网页特效 - 修改浏览器标题</nav>
11 </header>
12 <!-- 添加文档主体内容 -->
13 <div id="id-div-center" style="">
14     新的浏览器标题：
15     <input type="text" id="id-input-title" value=""/>
16     <input type="button" value="修改浏览器标题"
                        onclick="modifyTitle('id-input-title');"/>
17 </div>
18 </body>
19 <script type="text/javascript">
20     /**
21      * modify browser title
22      * @param id
23      */
24     function modifyTitle(id) {
25         var new_title = document.getElementById(id).value;
26         document.title = new_title;
27     }
28 </script>
29 </html>
```

关于【代码 9-22】的说明：

- 第 15 行代码通过<input>标签元素定义了一个文本框，用于填写新的（打算修改的）浏览器标题。
- 第 16 行代码通过<input>标签元素定义了一个按钮，并定义了单击 onclick 事件处理方法（modifyTitle()），用于执行修改浏览器标题的操作。
- 第 24～27 行代码是 modifyTitle()事件方法的实现过程，其中第 26 行代码通过修改 Document 对象的 title 属性来完成修改浏览器标题的操作。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 9.19 所示。

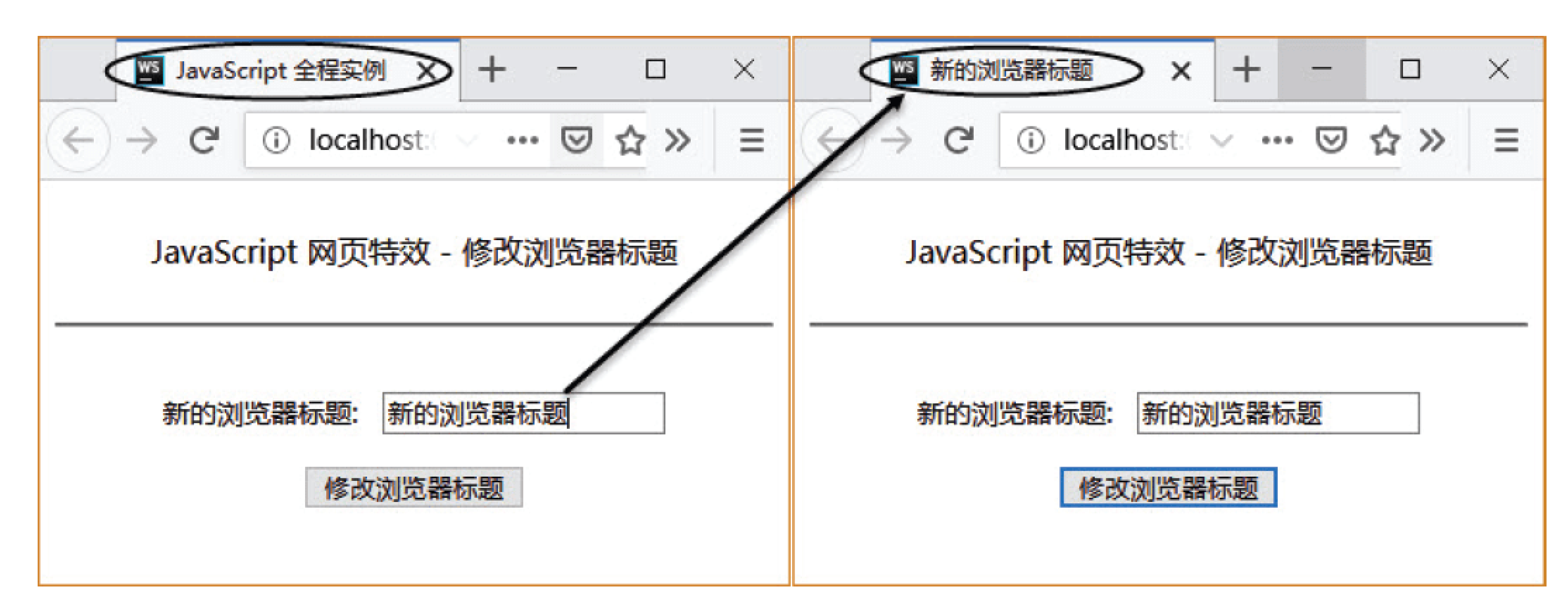


图 9.19 JavaScript 实现屏蔽键盘功能键

如图 9.19 所示，左边页面中的浏览器标题为初始定义的，右边页面中的浏览器标题为用户修改的（“新的浏览器标题”）。

第 10 章　DIV+CSS 特效

本章介绍如何通过 JavaScript 来实现 DIV 与 CSS（层叠样式表）的特效，通过这些特效来丰富 HTML 页面的设计手段和展示效果。

10.1　DIV 与层叠样式表概述

CSS（Cascading Style Sheets）是指层叠样式表，是 W3C 推出的格式化网页内容的标准技术，用于定义 HTML 元素的显示形式。CSS 主要包括盒子模型、列表模块、超链接方式、语言模块、背景和边框、文字特效、多栏布局等模块。

DIV（DIVision）是用于层叠样式表中的定位技术，即为划分（设计人员更习惯称为“层”）。在 HTML DOM 中，DIV 元素是用来为大块（block-level）的内容提供结构和背景的容器。

对于 HTML 网页布局方式的演变而言，DIV+CSS 方式是最新、最流行的 Web 设计标准。该方式与传统通过表格（table）布局定位方式的显著不同就是可以实现网页页面内容与表现相分离。DIV+CSS 方式使得页面结构更加清晰、逻辑更加合理、代码也更加精简，这样就提高了页面的访问速度，并增强了用户体验效果。

通过 JavaScript 脚本语言，可以针对 DIV+CSS 进行编程设计，实现多种多样的页面特效。后面将为读者介绍一些 DIV+CSS 特效的代码实例。

10.2　同时改变多个 DOM 样式

前文中提到，DIV+CSS 方式的显著特点就是将网页内容与表现相分离。简单来讲，就是页面中可能有多个 DOM 内容是同一类型属性的，可以使用 DIV 元素设计在同一个“层”内，并使用相对应的 CSS 样式进行表现。这样，当我们修改 CSS 样式时，就会同时改变多个 DOM 内容的样式。

下面介绍一个通过 JavaScript 实现同时改变多个 DOM 样式的代码实例。

【代码 10-1】（详见源代码目录 ch10-js-html-divcss-dom.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
```

```
05     <title>JavaScript 全程实例</title>
06 </head>
07 <body>
08 <!-- 添加文档主体内容 -->
09 <header>
10     <nav>JavaScript 网页特效 - 同时改变多个 DOM 的样式</nav>
11 </header>
12 <!-- 添加文档主体内容 -->
13 <div id="id-div-center" style="">
14     <p>第一段内容...</p>
15     <p>第二段内容...</p>
16     <p>第三段内容...</p>
17     <input type="button" onclick="changeFont(20);" value="大字体"/>
18     <input type="button" onclick="changeFont(14);" value="正常字体"/>
19     <input type="button" onclick="changeFont(8);" value="小字体"/>
20 </div>
21 </body>
22 <script type="text/javascript">
23     function changeFont(size) {
24         var fontSize = size + 'px';
25         var ps = document.getElementsByTagName('p');
26         for (var i = 0; i < ps.length; i++) {
27             ps[i].style.fontSize = fontSize;
28         }
29     }
30 </script>
31 </html>
```

关于【代码 10-1】的说明：

- 第 14～16 行代码通过<p>标签元素定义了一组段落，后面将通过 JavaScript 脚本同时改变这一组段落文本的字体大小。
- 第 17～19 行代码通过<input>标签元素定义了一组按钮，并定义了单击 onclick 事件方法（changeFont()），用于执行改变字体大小的操作。
- 第 23～29 行代码是 changeFont()方法的实现过程。其中，第 25 行代码先获取了全部<p>标签元素的对象集合，第 26～28 行代码通过 for 循环迭代语句遍历全部<p>标签元素，通过第 27 行代码修改 Style 对象的 fontSize 属性来改变<p>标签元素内容的字体大小。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 10.1 所示。

如图 10.1 箭头和标识所示，左边页面中为初始正常字体的效果，右边页面中为改变到大字体的效果。读者可以自行测试一下改变为小字体的效果。

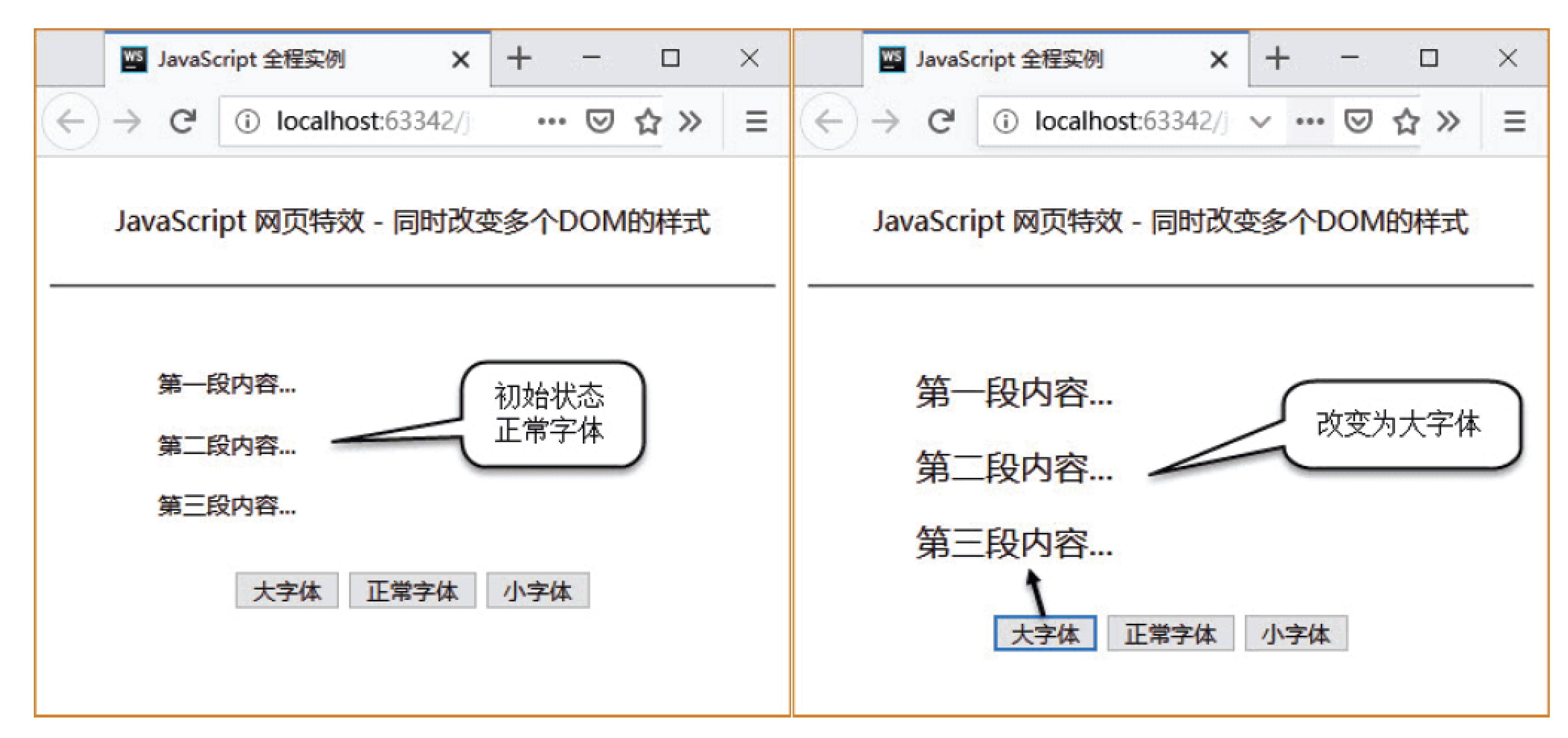


图 10.1　JavaScript 实现同时改变多个 DOM 样式（一）

读者可以注意到，【代码 10-1】是通过逐个修改 DOM 对象的 Style 属性来完成改变样式的。其实，还有一种更灵活的方式，就是通过修改样式类名（className）实现统一改变相关 DOM 对象的样式。下面看一个具体的 JavaScript 代码实例。

【代码 10-2】（详见源代码目录 ch10-js-html-divcss-dom-className.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<style type="text/css">
    .content-small {
        font-style: italic;
        font-size: small;
    }
    .content-medium {
        font-style: normal;
        font-size: medium;
    }
    .content-large {
        font-style: italic;
        font-size: large;
    }
</style>
<body>
```

```
22 <!-- 添加文档主体内容 -->
23 <div id="id-div-center" style="">
24     <p class="content-medium">这里是第一段...</p>
25     <p class="content-medium">这里是第二段...</p>
26     <p class="content-medium">这里是第三段...</p>
27     <input type="button" onclick="changeClass('content-small');"
                              value="Small 风格"/>
28     <input type="button" onclick="changeClass('content-medium');"
                              value="Medium 风格"/>
29     <input type="button" onclick="changeClass('content-large');"
                              value="Large 风格"/>
30 </div>
31 </body>
32 <script type="text/javascript">
33     function changeClass(s) {
34         var contents = document.getElementsByTagName('p');
35         for (var i = 0; i < contents.length; i++) {
36             contents[i].className = s;
37         }
38     }
39 </script>
40 </html>
```

关于【代码 10-2】的说明：

- 这段代码比较特殊的地方就是，第 08 ~ 11 行（.content-small）、第 12 ~ 15 行（.content-medium）和第 16 ~ 19 行（.content-large）代码定义的 CSS 样式代码，分别定义了三种风格样式的文本字体。
- 第 24 ~ 26 行代码通过<p>标签元素定义了一组段落，注意增加了 class 类名属性的定义（"content-medium"）。
- 第 27 ~ 29 行代码通过<input>标签元素定义了一组按钮，并定义了单击 onclick 事件方法（changeClass()），用于执行改变字体大小的操作。
- 第 33 ~ 38 行代码是 changeClass()方法的实现过程。其中，第 34 行代码先获取了全部<p>标签元素的对象集合，然后第 35 ~ 37 行代码通过 for 循环迭代语句遍历全部<p>标签元素，通过第 36 行代码修改 className 属性值来改变<p>标签元素内容的字体样式。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 10.2 所示。

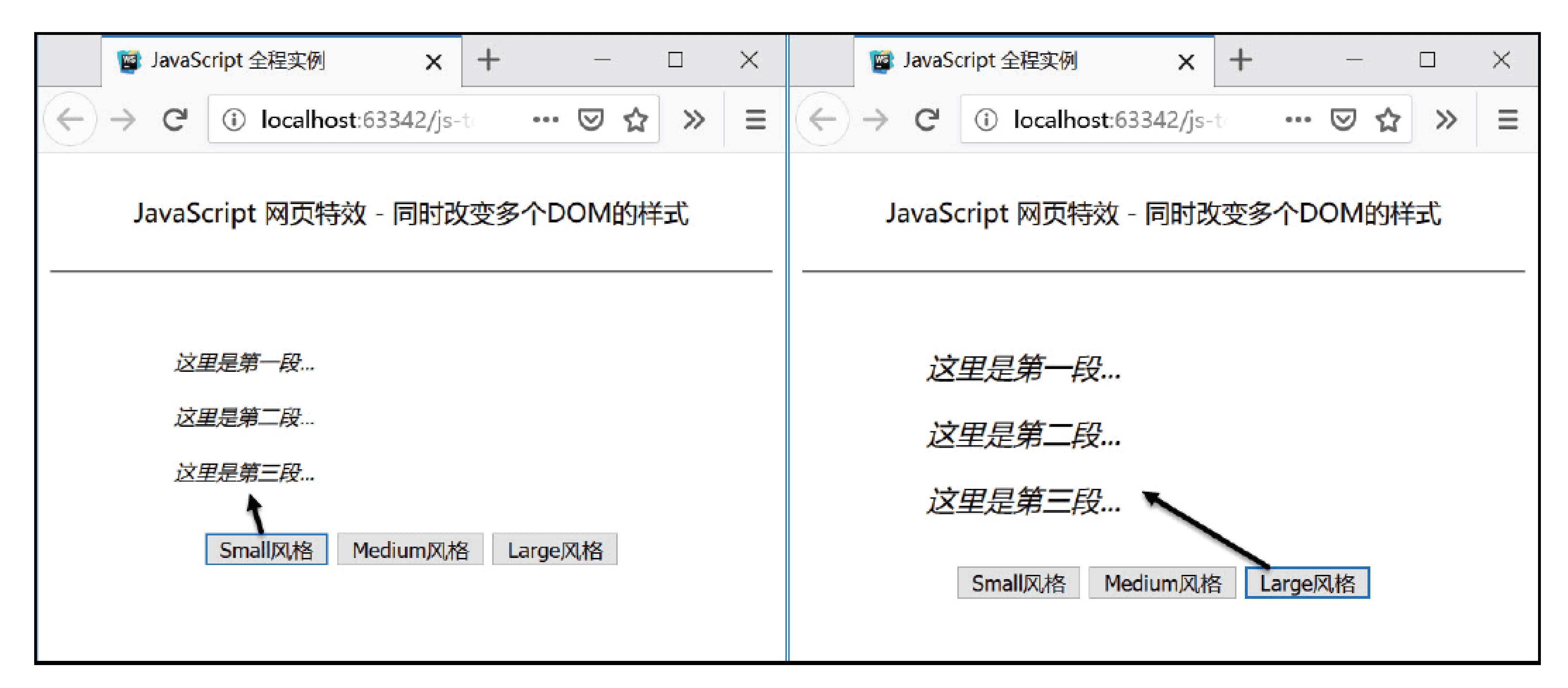


图 10.2　JavaScript 实现同时改变多个 DOM 样式（二）

10.3　弹　出　层

所谓弹出层，就是指通过 DIV+CSS 来模拟实现一个弹出窗口（这并不是一个真正意义的传统 Window 对象弹出窗口）。弹出层的应用场景非常多，可以模拟消息框、警告框、确认框等。而且由于弹出层的可编程性很强，可以设计出来各种各样的风格效果，因此在很多流行的前端框架中已经用其取代了传统弹出窗口的功能。

下面介绍如何通过 JavaScript 脚本语言来实现一个很简单的、只带关闭按钮的弹出层的代码实例。

【代码 10-3】（详见源代码目录 ch10-js-html-divcss-popup.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<style type="text/css">
    div#id-div-popup {
        display: none;
        width: 300px;
        height: 100px;
        border: 1px solid #ccc;
        text-align: center;
        font: normal 14px/1.2em "Microsoft Yahei", arial, sans-serif;
    }
</style>
```

```
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript DIV+CSS 特效 - 弹出层</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-popup">
    <p><input type="button" id='id-input-popupClose' value='close'/></p>
    <p>这是一个弹出层!</p>
</div>
<input type="button" id='id-input-popupOpen' value='弹出'/>
</body>
<script type="text/javascript">
    window.onload = function () {
        function setCss(_this, cssOption) { // TODO: 设置元素样式
            // TODO: 判断节点类型
            if (!_this || _this.nodeType === 3 || _this.nodeType === 8
                    || !_this.style) {
                return;
            }
            for (var cs in cssOption) {
                _this.style[cs] = cssOption[cs];
            }
        }
        function setPopup(e, openPop, closePop) {
            setCss(e, { // TODO: 初始化样式
                "position": "absolute",
                "zIndex": 100,
                "backgroundColor": "#eee"
            });
            openPop.onclick = function () {
                e.style.display = "block";
                setCss(e, {
                    "top": "50%",
                    "left": "50%",
                    "marginLeft": -e.offsetWidth / 2 + "px"
                });
            };
            closePop.onclick = function () {
                e.style.display = "none";
            };
        }
        setPopup(
```

```
59                document.getElementById("id-div-popup"),
60                document.getElementById("id-input-popupOpen"),
61                document.getElementById("id-input-popupClose")
62            );
63        };
64  </script>
65  </html>
```

关于【代码 10-3】的说明：

- 第 23～26 行代码通过<div>标签元素定义了一个层，其中第 24 行代码通过<input id='id-input-popupClose'>标签元素定义了一个关闭（close）按钮、第 25 行代码定义了一段文本，我们用这个设计来模拟一个弹出层。
- 第 27 行代码通过<input id='id-input-popupOpen'>标签元素定义了一个按钮，用于激活弹出层。
- 第 31～39 行代码的自定义方法 setCss()用于设置元素样式，其中第 36～38 行代码通过 for 循环迭代语句将全部样式依次设置到元素中。
- 第 40～57 行代码的自定义方法 setPopup()用于设置弹出层，具体内容如下。
  - 第 41～45 行代码调用 setCss()方法，将元素的 position 属性修改为 absolute（绝对定位），这样就可以随意在整个页面中进行定位了。
  - 第 46～52 行代码绑定了<input id='id-input-popupOpen'>按钮的单击 click 事件方法。其中，第 47 行代码将弹出层的 display 属性值改为 block（可见状态）；第 48～52 行代码调用 setCss()方法设置弹出层的坐标为窗口的中间位置。
  - 第 54～56 行代码绑定了<input id='id-input-popupClose'>按钮的单击 click 事件方法，用于关闭弹出层。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 10.3 所示。

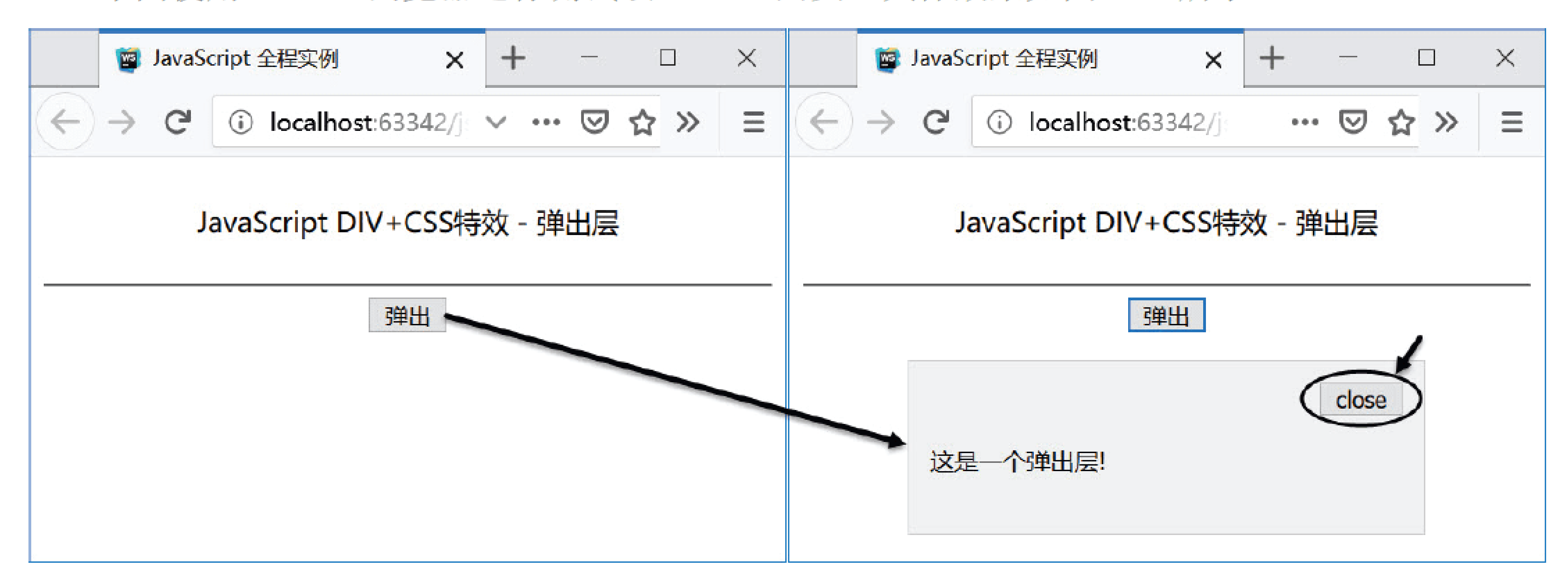


图 10.3　JavaScript 实现 DIV+CSS 弹出层

如图 10.3 中箭头和标识所示，单击左边页面中的“弹出”按钮，弹出层的效果如右边页面所示；在右边页面中单击“close”按钮，则会关闭该弹出层。

10.4 用层模拟确认框

在 10.3 节实现的弹出层其实更像一个简单的弹出式消息框。前文中我们也提到了，用弹出层还可以模拟出稍微复杂一些的确认框效果。下面介绍一个通过 JavaScript 实现用层模拟出确认框效果的代码实例。

【代码 10-4】（详见源代码目录 ch10-js-html-divcss-prompt.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<style type="text/css">
    div#id-div-prompt {
        display: none;
        width: 300px;
        height: 100px;
        border: 1px solid #ccc;
        text-align: center;
        font: normal 14px/1.2em "Microsoft Yahei", arial, sans-serif;
    }
</style>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript DIV+CSS 特效 - 用层模拟确认框</nav>
</header>
<!-- 添加文档主体内容 -->
<div id='id-div-prompt' style="">
    <p>用层模拟的确认框，请用户测试!</p>
    <p>
        <input type="button" id='id-prompt-ok' value='确认'/>
        <input type="button" id='id-prompt-cancel' value='取消'/>
    </p>
</div>
<input type="button" id='id-prompt-open' value='弹出确认框'/>
</body>
<script type="text/javascript">
    window.onload = function () {
```

```
    function div_prompt(options) {
        var e = options.prompt, // TODO: 获取元素
            setCss = function (_this, cssOption) {  // TODO: 设置元素样式
                // TODO: 判断节点类型
                if (!_this || _this.nodeType === 3 ||
                        _this.nodeType === 8 || !_this.style){
                    return;
                }
                for (var cs in cssOption) {
                    _this.style[cs] = cssOption[cs];
                }
                return _this;
            };
        setCss(e, { // TODO: 初始化基本样式
            "position": "absolute",
            "zIndex": 100,
            "top":((document.body.scrollTop || document.
                    documentElement.scrollTop) + window.screen.
                    availHeight / 2 - e.offsetHeight) + "px",
            "backgroundColor": "#eee"
        });
        options.promptOpen.onclick = function () {  // TODO: 打开确认按钮
            e.style.display = "block";
            setCss(e, { // TODO: 设置位置
                "left": "50%",
                "top": "50%",
                "marginLeft": -e.offsetWidth / 2 + "px"
            });
        };
        options.promptOk.onclick = function () {    // TODO: 确认按钮
            e.style.display = "none";   // TODO: 隐藏层
            if (options.okCallBack) options.okCallBack();
        };
        options.promptCancel.onclick = function () {   // TODO: 取消按钮
            e.style.display = "none";   // TODO: 隐藏层
            if (options.cancelCallBack) options.cancelCallBack();
        };
    }
    div_prompt({
        "prompt": document.getElementById("id-div-prompt"),
```

```
71              "promptOpen": document.getElementById("id-prompt-open"),
72              "promptOk": document.getElementById("id-prompt-ok"),
73              "okCallBack": function () {
74                  alert("确认的回调函数！");
75              },
76              "promptCancel": document.getElementById("id-prompt-cancel"),
77              "cancelCallBack": function () {
78                  alert("取消的回调函数！");
79              }
80          });
81      };
82  </script>
83  </html>
```

关于【代码 10-4】的说明：

- 第 23～29 行代码通过<div>标签元素定义了一个层。其中，第 24 行代码定义了一段文本，第 26 行代码通过<input id='id-prompt-ok'>标签元素定义了一个确认按钮，第 27 行代码通过<input id='id-prompt-cancel'>标签元素定义了一个取消按钮，这样用层设计来模拟出一个确认框。
- 第 30 行代码通过<input id='id-prompt-open'>标签元素定义了一个按钮，用于激活确认框。
- 第 34～68 行代码的自定义方法 div_prompt()实现用层模拟出来的确认框，具体内容如下。
  - 第 36～45 行代码的自定义方法 setCss()用于设置元素样式，其中第 41～43 行代码通过 for 循环迭代语句将全部样式依次设置到元素中。
  - 第 46～51 行代码调用 setCss()方法，将元素的 position 属性修改为 absolute（绝对定位），这样就可以随意在整个页面中进行定位了。
  - 第 52～59 行代码绑定了<input id='id-prompt-open'>按钮的单击 click 事件方法，第 53 行代码将弹出层的 display 属性值改为 block（可见状态），第 54～58 行代码调用 setCss()方法设置弹出层的坐标为窗口的中间位置。
  - 第 60～63 行代码绑定了<input id='id-prompt-ok'>按钮的单击 click 事件方法，用于执行确认操作。
  - 第 64～67 行代码绑定了<input id='id-prompt-cancel'>按钮的单击 click 事件方法，用于执行取消操作。
- 第 69～80 行代码通过调用 div_prompt()自定义方法并设定相关参数，实现了用层模拟出来的确认框。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 10.4 所示。

如图 10.4 中箭头所示，页面中分别演示了用层模拟确认框的效果，以及用户单击“确认”和“取消”后执行回调函数的效果。

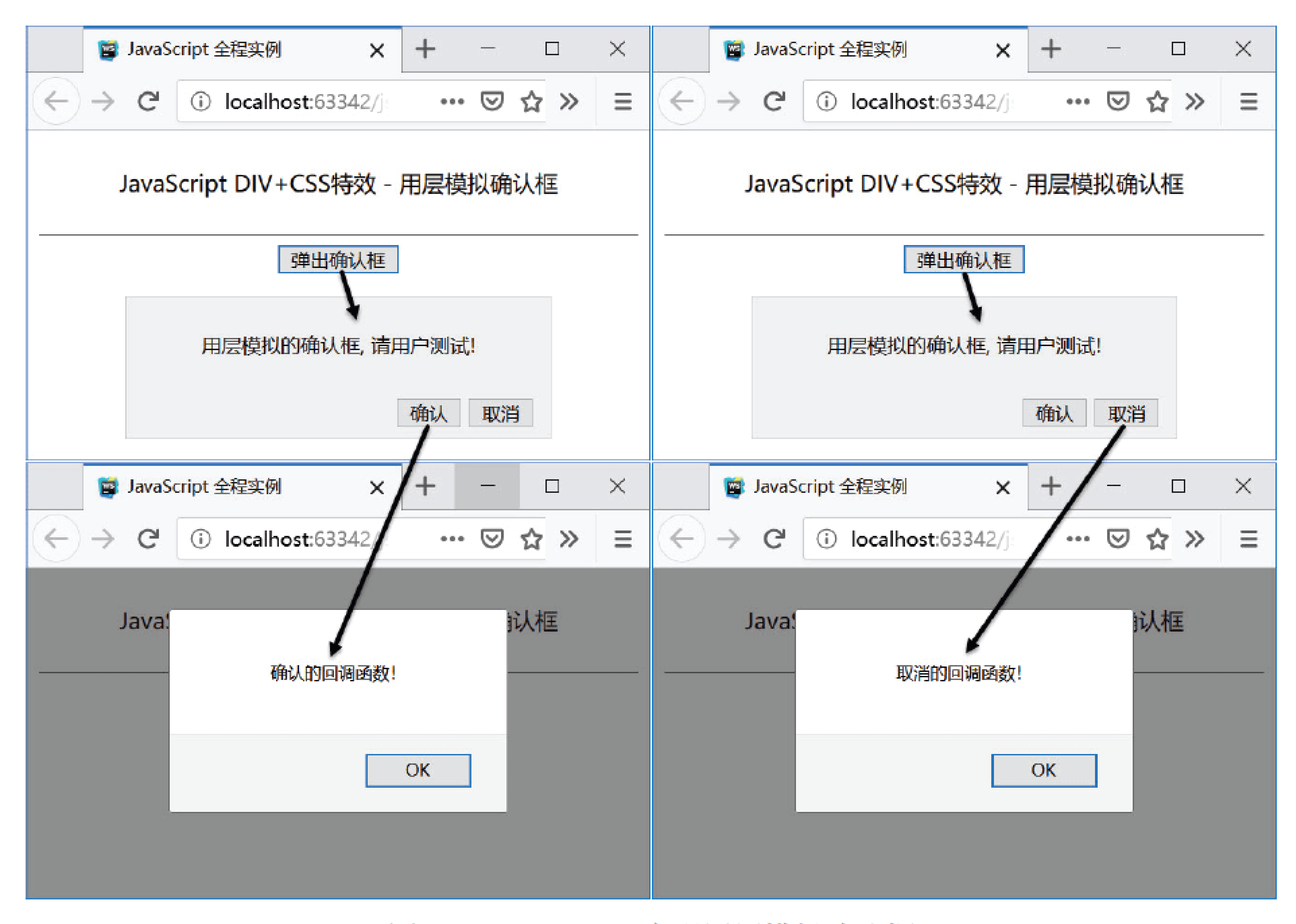


图 10.4　JavaScript 实现用层模拟确认框

10.5　隐　藏　层

在 HTML 网页设计中，对于层（DIV）的使用是非常灵活的。在很多设计场景下，需要层在可见和隐藏两种状态下反复切换来满足实际需求，这就是隐藏层的舞台了。一般，隐藏层是通过设定 CSS 样式的 Display 属性（none|block）实现的。下面介绍一个通过 JavaScript 实现隐藏层的代码实例。

【代码 10-5】（详见源代码目录 ch10-js-html-divcss-hidden.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript DIV+CSS 特效 - 隐藏层</nav>
</header>
<!-- 添加文档主体内容 -->
```

```
13 <div id="id-div-hidden">
14     <p>这是一个隐藏层!</p>
15     <p>请用户切换“隐藏/显示”效果!</p>
16 </div>
17 <input type="button" id='id-input-hidden' value='隐藏上面的层'/>
18 <input type="button" id='id-input-show' value='显示隐藏层'/>
19 </body>
20 <script type="text/javascript">
21     window.onload = function () {
22         document.getElementById("id-input-hidden").onclick = function () {
23             document.getElementById("id-div-hidden").style.
                                           display = "none";     // 隐藏元素
24         };
25         document.getElementById("id-input-show").onclick = function () {
26             document.getElementById("id-div-hidden").style.
                                           display = "block";    // 显示元素
27         };
28     };
29 </script>
30 </html>
```

关于【代码 10-5】的说明：

- 第 13 ~ 16 行代码通过<div id="id-div-hidden">标签元素定义了一个层，并定义了一些文本，我们用这个设计来模拟一个隐藏层。
- 第 17 行和第 18 行代码通过<input>标签元素定义了两个按钮，分别用于执行隐藏和显示上面第 13 ~ 16 行代码所定义隐藏层的操作。
- 第 23 行和第 26 行代码分别通过设定层（<div id="id-div-hidden">）的 Display 属性值（"none"表示层不可见；"block"表示层为块级元素，也就是可见的）来实现隐藏层的功能。

10.6 可拖动的层

在前面介绍的几个代码实例中，层（DIV）都是位置固定的，也就是无法通过鼠标进行拖动的。而在实际项目需求中，层（DIV）一般都是可以拖动的，比如模拟出来的弹出层是可以满足用户拖动改变位置的。JavaScript 实现可拖动的层，需要对鼠标键的按下（MouseDown）、移动（MouseMove）和松开（MouseUp）事件进行编程，并根据鼠标位置的变化计算出可拖动层的新位置。下面介绍一个通过 JavaScript 实现可拖动层的代码实例。

【代码 10-6】（详见源代码目录 ch10-js-html-divcss-draggable.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript DIV+CSS 特效 - 可拖动的层</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-box" style="">
    <h3 align="right" id="id-box-drag" style="">
        <span style="float:left; color:white;">标题栏(鼠标拖动)</span>
    </h3>
    <div id="id-box-msg">这是一个可拖动的层!</div>
    <div id="id-box-pos"></div>
</div>
</body>
<script type="text/javascript">
    var isDrag = false; // TODO: 是否正在移动的标志变量
    var l = 0;          // TODO: left 变量
    var t = 0;          // TODO: top 变量
    var x, y;
    window.onload = function (ev) {
        var boxDrag = document.getElementById('id-box-drag');
        boxDrag.onmousedown = mouseDown;    // TODO: 设置 mousedown 事件
        boxDrag.onmousemove = mouseMove;    // TODO: 设置 mousemove 事件
        boxDrag.onmouseup = mouseUp;        // TODO: 设置 mouseup 事件
        var initPos = "X pos : 200 , Y pos : 100";
        document.getElementById('id-box-pos').innerText = initPos;
    };
    /**
     * 定义 box 的鼠标按下事件
     */
    function mouseDown(event) {
        var e = event;
        x = e.clientX; // TODO: 得到事件发生的 x 坐标
        y = e.clientY; // TODO: 得到事件发生的 y 坐标
        // TODO: 得到 box 左上角的坐标
```

```
42          l = document.getElementById("id-div-box").offsetLeft;
43          t = document.getElementById("id-div-box").offsetTop;
44          isDrag = true;
45       }
46       /**
47        * 定义 box 的鼠标移动事件
48        */
49       function mouseMove(event) {
50          if (isDrag) {
51             var e = event;
52             var box = document.getElementById("id-div-box");
53             var newLeft = l + e.clientX - x;    // TODO: 计算新的 left 的值
54             var newTop = t + e.clientY - y;     // TODO: 计算新的 top 的值
55             var newPos = "X pos : " + newLeft + ", Y pos : " + newTop;
56             document.getElementById('id-box-pos').innerText = newPos;
57             box.style.left = newLeft + "px";
                                              // TODO: 设置新的 left 值，带上单位
58             box.style.top = newTop + "px"; // TODO: 设置新的 top 值，带上单位
59          }
60       }
61       /**
62        * 定义 box 的鼠标按下释放事件
63        */
64       function mouseUp(event) {
65          if (isDrag) {
66             isDrag = false;
67          }
68       }
69    </script>
70    </html>
```

关于【代码 10-6】的说明：

- 第 13 ~ 19 行代码通过<div id="id-div-box">标签元素定义了一个层。其中，第 14 ~ 16 行代码通过<h3>标签元素定义了一个标题栏（用于鼠标拖动），第 17 ~ 18 行代码分别定义了一行文本和一个空行用于显示动态坐标。我们用这个设计来模拟一个可拖动的层。
- 第 22 行代码定义了一个变量（isDrag），是用于表示当前状态是否为鼠标正在拖动的一个标志量。
- 第 37 ~ 45 行代码、第 49 ~ 60 行代码和第 64 ~ 68 行代码分别绑定了可拖动层（<div id="id-div-box">）的鼠标按下、鼠标移动和鼠标按下释放这三个事件方法，通过计算可拖动层的坐标位置来实现层的拖动效果。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 10.5 所示。

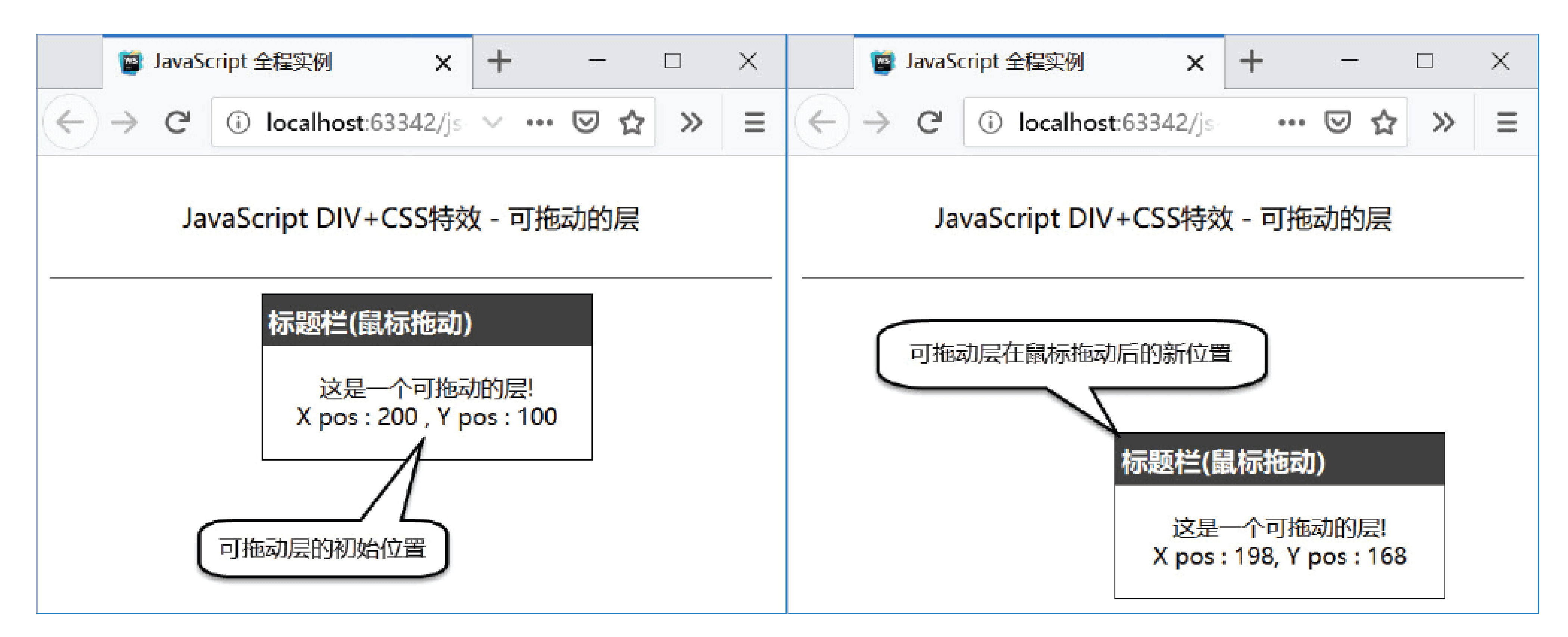


图 10.5　JavaScript 实现可拖动的层

如图 10.5 中的标识所示，左边页面中显示了可拖动层的初始位置，右边页面中显示了可拖动层在经过鼠标拖动后的新位置。

10.7　遮罩层效果

所谓遮罩层，就是为了限制用户操作页面中的元素而遮罩住全部页面或局部页面的效果。例如，在需要实名认证的网页中会弹出一个提示框，要求用户填写认证信息，如果用户不填写认证信息就通过遮罩的形式限制用户操作页面内的其他元素。下面介绍一个通过 JavaScript 实现遮罩层效果的代码实例。

【代码 10-7】（详见源代码目录 ch10-js-html-divcss-masklayer.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<style type="text/css">
    div#id-div-layer {
        position: relative;
        width: 320px;
        height: 160px;
        border: 1px solid gray;
        margin: 32px auto;
        padding: 2px;
        font: normal 14px/1.2em "Microsoft Yahei", arial, sans-serif;
    }
</style>
```

```
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript DIV+CSS 特效 - 遮罩层效果</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-layer" style="">
    <p>测试遮罩层效果:</p>
    <p><input type="text"></p>
    <p><input type="button" value="提交"></p>
    <div id='id-div-maskLayer' style="display: none"></div>
</div>
<input type="button" value='显示遮罩层' id='id-input-showMaskLayer'/>
<input type="button" value='取消遮罩层' id='id-input-cancelMaskLayer'/>
<input type="button" value='遮罩全部页面' id='id-input-bodyMaskLayer'/>
<div id='id-div-bodyMaskLayer' style="display: none"></div>
</body>
<script type="text/javascript">
    window.onload = function () {
        function setCss(_this, cssOption) { // TODO: 设置元素样式
            // TODO: 判断节点类型
            if (!_this || _this.nodeType === 3 || _this.nodeType === 8
                        || !_this.style) {
                return;
            }
            for (var cs in cssOption) { // TODO: 遍历设置样式
                _this.style[cs] = cssOption[cs];
            }
            return _this;
        }
        document.getElementById("id-input-showMaskLayer").
                                            onclick = function () {
            var divLayer = document.getElementById("id-div-layer"),
                maskLayer = document.getElementById("id-div-maskLayer");
            setCss(maskLayer, { // TODO: 初始化遮罩的样式
                "position": "absolute",
                "display": "block",
                "left": "0px",
                "top": "0px",
                "zIndex": 1000,
                "backgroundColor": "#ccc",
                "height": divLayer.scrollHeight + "px",
```

```
                "width": divLayer.offsetWidth + "px",
                /*兼容各种浏览器的透明层效果*/
                "filter": "alpha(Opacity=60)",
                "opacity": "0.6",
                "-moz-opacity": "0.6",
                "filter": "progid:DXImageTransform.Microsoft.Alpha(Opacity=60)",
                "-MS-filter": "progid:DXImageTransform.Microsoft.Alpha(Opacity=60)"
            });
        };
        document.getElementById("id-input-cancelMaskLayer").onclick = function () {
            var divLayer = document.getElementById("id-div-layer"),
                maskLayer = document.getElementById("id-div-maskLayer");
            setCss(maskLayer, { // TODO: 初始化遮罩的样式
                "display": "none"
            });
        };
        document.getElementById("id-input-bodyMaskLayer").onclick = function () {
            var body = document.body.parentNode,
                maskLayer = document.getElementById("id-div-bodyMaskLayer");
            setCss(maskLayer, {            // TODO: 初始化遮罩的样式
                "position": "absolute",
                "display": "block",
                "left": "0px",
                "top": "0px",
                "zIndex": 1000,
                "backgroundColor": "#ccc",
                "height": body.scrollHeight + "px",
                "width": body.offsetWidth + "px",
                /*兼容各种浏览器的透明层效果*/
                "filter": "alpha(Opacity=60)",
                "opacity": "0.6",
                "-moz-opacity": "0.6",
                "filter": "progid:DXImageTransform.Microsoft.Alpha(Opacity=60)",
                "-MS-filter": "progid:DXImageTransform.Microsoft.Alpha(Opacity=60)"
            });
```

```
93          };
94      };
95  </script>
96  </html>
```

关于【代码 10-7】的说明：

- 第 24 ~ 29 行代码通过<div id="id-div-layer">标签元素定义了一个层，该层的 CSS 样式定义见第 08 ~ 16 行代码。其中，第 25 ~ 27 行代码定义了一些可操作的元素，目的是模拟出一个表单内容；第 28 行代码通过<div id='id-div-maskLayer'>标签元素定义了一个遮罩层，这里注意一下 Style 样式中 display 属性（none）的使用。我们通过这样的设计来实现遮罩层的效果。
- 第 33 行代码通过<div id='id-div-bodyMaskLayer'>标签元素定义了另一个遮罩层，与前一个遮罩层（<div id='id-div-maskLayer'>）不同的是，通过这个遮罩层可以实现整个页面被遮罩的效果。
- 第 37 ~ 46 行代码的自定义方法 setCss()用于设置元素样式，其中第 42 ~ 44 行代码通过 for 循环迭代语句将全部样式依次设置到元素中。
- 第 47 ~ 66 行代码定义的事件处理方法实现了层（<div id="id-div-layer">）的遮罩效果，具体内容如下。
  - ➢ 第 52 行代码将遮罩层（<div id='id-div-maskLayer'>）的 display 属性值改为 block（可见状态）。
  - ➢ 第 57 ~ 58 行代码根据层（<div id="id-div-layer">）的宽高尺寸重新计算了遮罩层（<div id='id-div-maskLayer'>）的宽高尺寸。
  - ➢ 第 60 ~ 64 行代码定义了兼容各种浏览器的透明层效果。
- 第 74 ~ 94 行代码定义的事件处理方法实现了整个页面的遮罩效果。注意，第 75 行代码获取的是 body 对象的父元素节点。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 10.6 和图 10.7 所示。

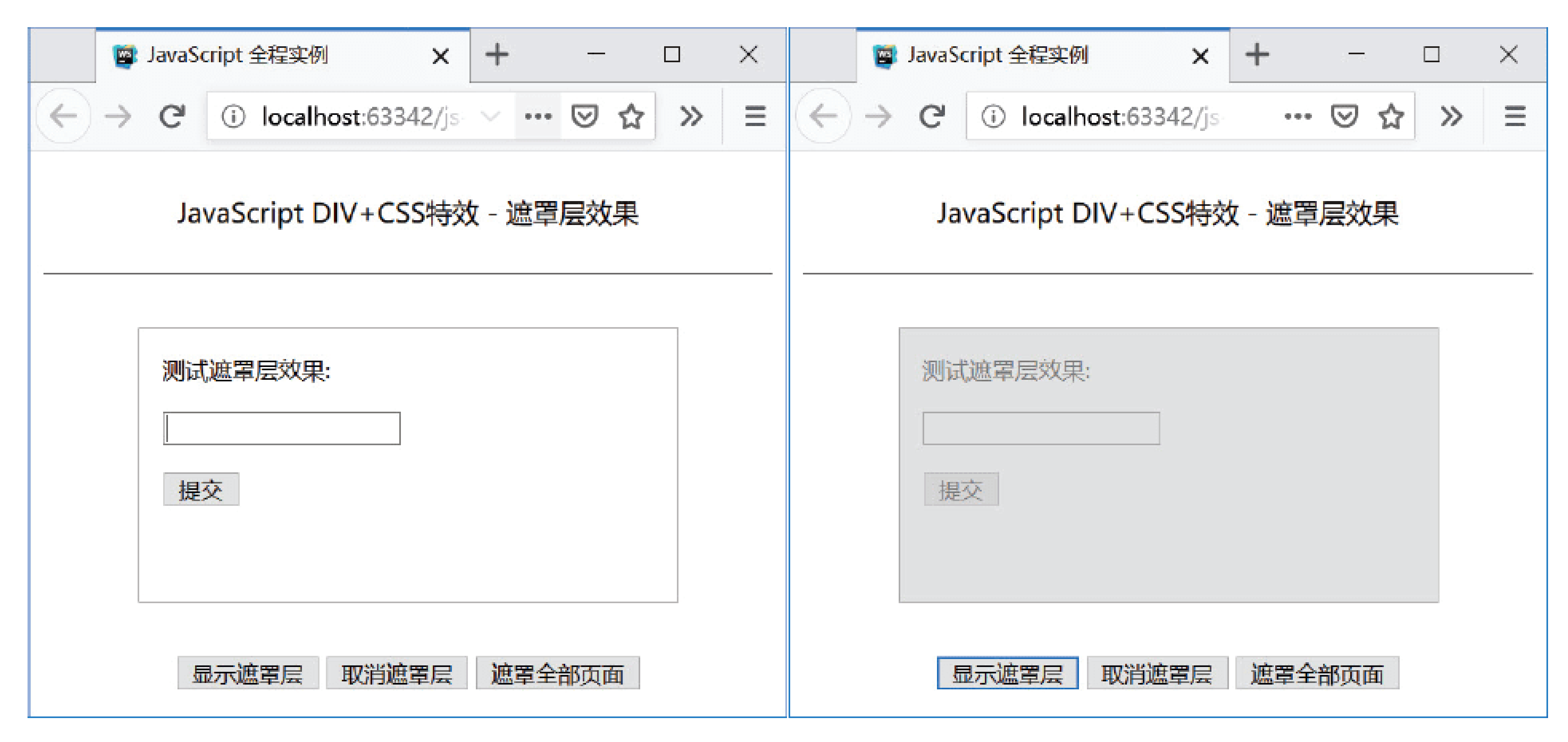


图 10.6　JavaScript 实现遮罩层效果（一）

如图 10.6 所示，左边页面中显示了方框区域未被遮罩的效果，右边页面中显示了方框区域被遮罩的效果。

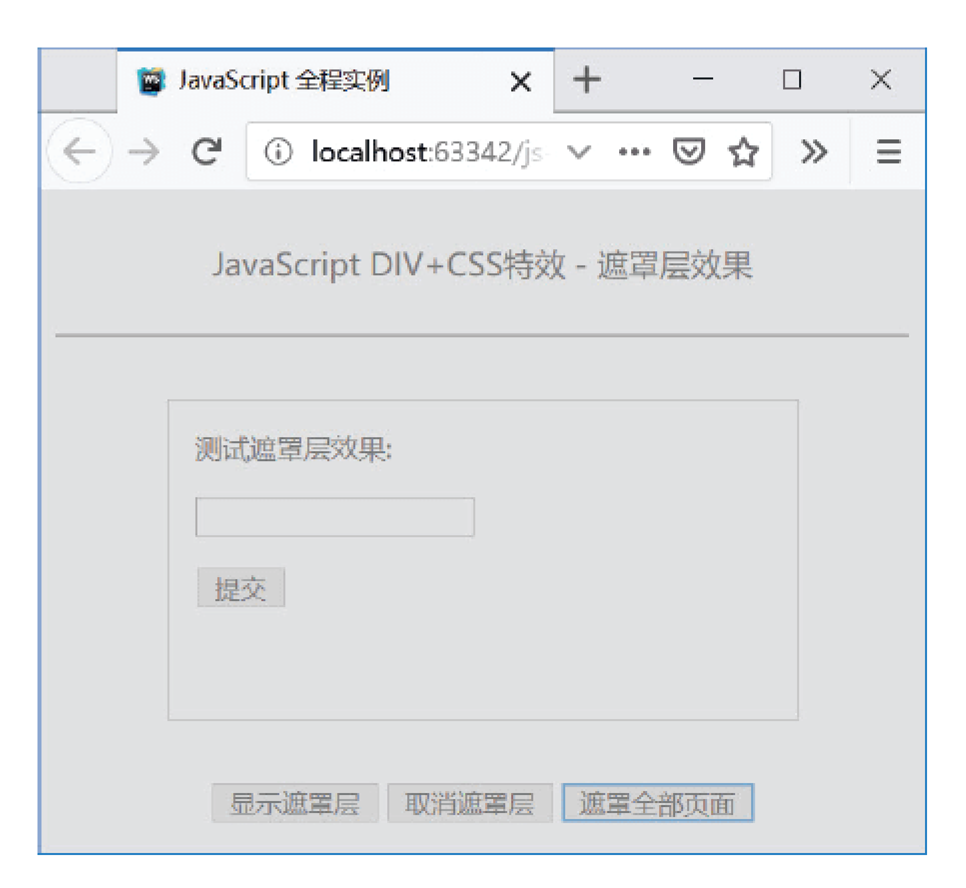


图 10.7 JavaScript 实现遮罩层效果（二）

如图 10.7 所示，该页面显示了整个页面被遮罩层所遮罩的效果。

10.8 Tab 选项卡

Tab 选项卡可以帮助网页用户自由切换自己想要的内容及模块。例如，购物网站有很多产品，每个产品都有自己的种类，如果将所有的产品名称都显示出来是不友好的。这时就需要将产品按类别处理，每个类别之间使用 Tab 选项卡进行切换，这样不仅可以提高浏览速度，还增强了用户交互体验效果。下面介绍一个通过 JavaScript 实现 Tab 选项卡的代码实例。

【代码 10-8】（详见源代码目录 ch10-js-html-divcss-tab.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<style type="text/css">
    .tabSwitchParent {
        position: relative;
    }
    .tabSwitchParent div {
        position: relative;
        float: left;
    }
    .tabSwitchTarget {
        display: none;
        left: 0px;
```

```
            top: 0px;
            z-index: 1;
            width: 320px;
            height: 64px;
            padding: 32px 8px 16px 8px;
            border: 1px solid #aaa;
            color: #333;
            background-color: #eee;
        }
        #tabSwitch {
            position: relative;
            float: left;
            z-index: 2;
            margin: 0px;
            top: 1px;
            bottom: 0px;
            font-size: 14px;
            text-align: center;
            cursor: pointer;
        }
        #tabSwitch div {
            padding: 8px;
            border: 1px solid #ccc;
            border-bottom: none;
            background-color: #fff;
        }
        #tabSwitch div.on {
            border: 1px solid #aaa;
            border-bottom: none;
            background-color: #eee;
        }
</style>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript DIV+CSS 特效 - Tab 选项卡</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-center" style="">
    <div class="tabSwitchParent">
        <!--选项卡-->
        <div id='tabSwitch'>
```

```
            <div data-target='tabSwitch1' class="on">JavaScript</div>
            <div data-target='tabSwitch2'>jQuery</div>
            <div data-target='tabSwitch3'>NodeJS</div>
        </div>
        <!--内容-->
        <div class="tabSwitchTarget" style="display: block;"
                                                   id='tabSwitch1'>
            欢迎您学习《JavaScript 全程实例》
        </div>
        <!--内容-->
        <div class="tabSwitchTarget" style="" id='tabSwitch2'>
            欢迎您学习《jQuery 全程实例》
        </div>
        <!--内容-->
        <div class="tabSwitchTarget" style="" id='tabSwitch3'>
            欢迎您学习《NodeJS 全程实例》
        </div>
    </div>
</div>
</body>
<script type="text/javascript">
    window.onload = function () {
        function getTypeElement(es, type) {   //获取指定类型的节点
            var esLen = es.length,
                i = 0,
                eArr = [],
                esI = null;
            for (; i < esLen; i++) {
                esI = es[i];
                if (esI.nodeName.replace("#", "").
                                     toLocaleLowerCase() === type) {
                    eArr.push(esI);
                }
            }
            return eArr;
        }
        function tabSwitch(e) {
            var divs = getTypeElement(e.childNodes, "div"),
                l = divs.length,
                i = 0;
            for (; i < l; i++) {
                divs[i].onclick = function () {    //单击切换按钮
```

```
100                 for (var ii = 0; ii < l; ii++) {
101                 divs[ii].className = "";    //删除选项卡的边框
102              document.getElementById("tabSwitch" + (ii + 1)).style.
                                                  display = "none";
103                 }
104                 this.className = "on"; //设置当前元素的选中样式
105             document.getElementById(this.getAttribute("data-target")).
                                                style.display = "block";
106             }
107          }
108       }
109       tabSwitch(document.getElementById("tabSwitch")); //Tab 选项卡切换
110    };
111 </script>
112 </html>
```

关于【代码 10-8】的说明：

- 第 057 ~ 076 行代码通过一组<div>标签元素定义了一个 Tab 控件。其中，第 058 ~ 063 行代码定义了一组 Tab 选项卡，第 064 ~ 075 行代码定义了 Tab 选项卡所对应的 Tab 选项内容。
- 第 099 ~ 103 行代码为第 057 ~ 076 行代码定义的一组 Tab 选项卡分别绑定了单击 onclick 事件。当单击 Tab 选项卡标签时，修改被单击元素的样式为选中状态，其他元素修改为未选中状态，在被选中的元素节点上获取将要显示的选项内容 id，将所有的选项内容隐藏，即 “display="none"”，然后显示选取内容的对象，设置样式 “display="block"”。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 10.8 所示。

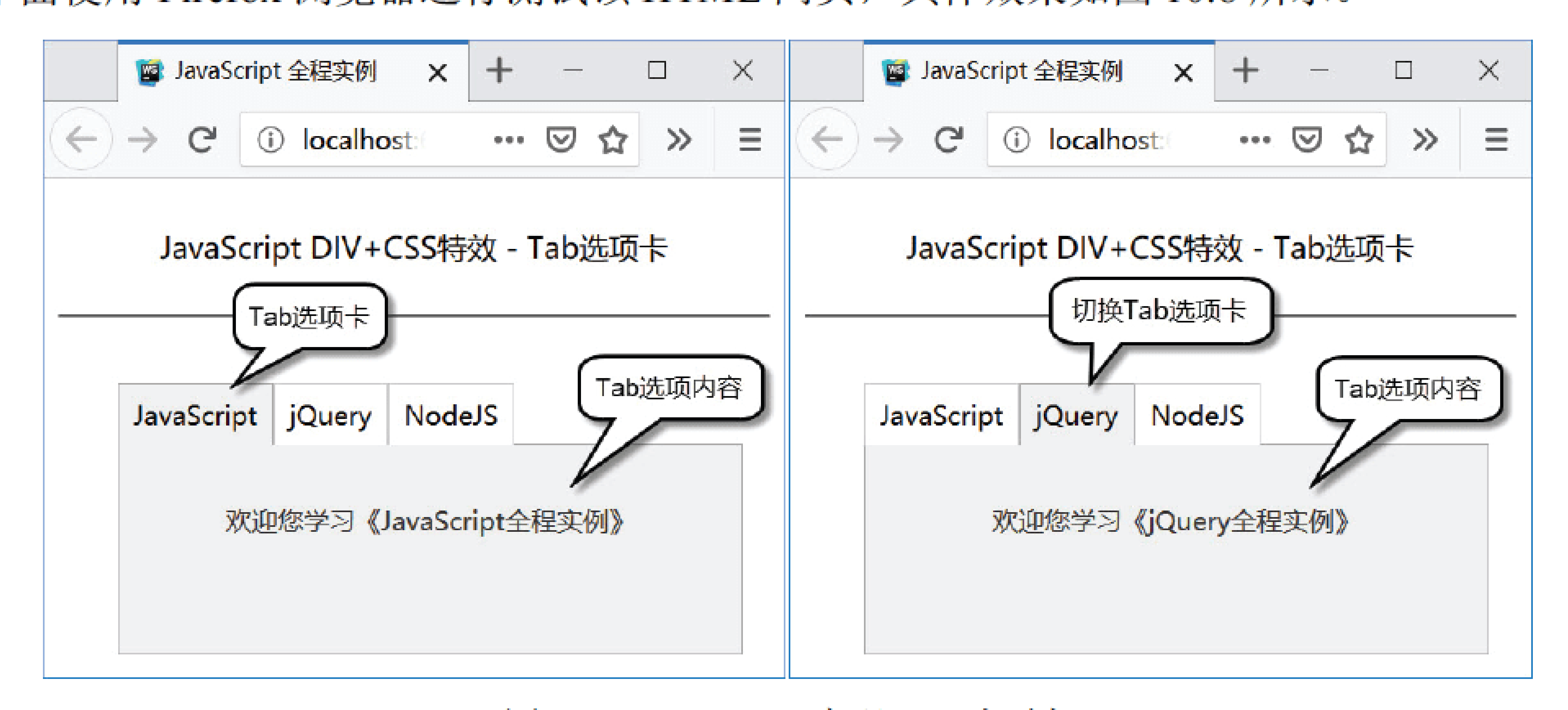


图 10.8　JavaScript 实现 Tab 选项卡

如图 10.8 中的标识所示，左边页面中显示了“JavaScript”Tab 选项卡的内容，右边页面中显示了切换到“jQuery”Tab 选项卡的内容。

第 11 章　Ajax 应用

本章介绍如何通过 Ajax 技术来实现 HTML 网页的应用开发，通过这些应用来丰富 HTML 页面的设计手段和展示效果。

11.1　Ajax 概述

Ajax（Asynchronous JavaScript and XML）即异步的 JavaScript 和 XML，是指一种创建交互式网页应用的前端开发技术。这里特别需要强调的是，Ajax 不是一门全新的编程语言，而是指一种前端开发新方法，这是大多数初学者特别容易混淆的概念。

Ajax 技术的特点就是前端页面与后台服务器进行数据交互的全新方式，通过核心的 XMLHttpRequest 对象在无须重新加载整个网页的情况下，可以实现独立更新网页中的局部内容。而传统的网页如果需要更新内容，则必须重载整个网页页面才可以实现。所以，Ajax 会被设计人员定义为“异步方式的 JavaScript 和 XML 技术”。

由于 Ajax 技术依赖于浏览器的 JavaScript 和 XML，因此浏览器的兼容性和支持程度也就和 JavaScript 运行时的性能一样重要了，还好这对于如今绝大多数的主流浏览器（Google Chrome、FireFox、Opera、Apple Safari、微软的 IE 系列和 Microsoft Edge 等）均不是问题了。Ajax 技术不需要额外安装任何浏览器插件，只要用户在浏览器中允许 JavaScript 脚本执行权限即可，这也是 Ajax 技术非常受欢迎的主要原因之一。

通过 Ajax 技术，可以实现很多种 Web 前端开发的特效。后面将为读者介绍一些常用的 Ajax 技术应用。

11.2　Ajax 基础

Ajax 技术的基础就是对核心对象 XMLHttpRequest 的使用，如今绝大多数的主流浏览器均支持 XMLHttpRequest 对象（不过早期微软的 IE5 和 IE6 浏览器使用的是 ActiveXObject 对象）。

对于 Google Chrome、FireFox、Opera、Apple Safari、微软 IE7+系列和 Microsoft Edge 等主流浏览器，内核中已经内建了 XMLHttpRequest 对象，因此创建 XMLHttpRequest 对象的语法可以写成如下形式：

```
var xhr = new XMLHttpRequest();
                    // TODO: Chrome、FireFox、Opera、Safari、IE7+、Edge
```

而对于老版本的 IE5 和 IE6，则使用 ActiveXObject 对象来创建：

```
var xhr = new ActiveXObject("Microsoft.XMLHTTP");  // TODO: IE5、IE6
```

综上所述，为了更好地编写浏览器兼容性代码（兼容绝大多数浏览器），可以先检查浏览器是否支持 XMLHttpRequest 对象，如果支持就创建 XMLHttpRequest 对象，如果不支持就创建 ActiveXObject 对象，具体的写法如下。

【代码 11-1】

```
01 var xhr = null;  // TODO: define XMLHttpRequest Object
02 if (window.XMLHttpRequest) {   // TODO: Chrome, Firefox, Opera, IE7+,
                                     Edge, etc.
03     xhr = new XMLHttpRequest();
04 } else if (window.ActiveXObject) {  // TODO: IE5, IE6
05     xhr = new ActiveXObject("Microsoft.XMLHTTP");
06 } else {
07     xhr = null;
08 }
```

关于【代码 11-1】的说明：

- 第 01 行代码定义了一个 XMLHttpRequest 对象实例（xhr）。
- 第 02 ~ 08 行代码通过 if 条件语句判断浏览器类型，并根据判断结果选择创建 XMLHttpRequest 对象的方式。

在 XMLHttpRequest 对象（xhr）创建成功后，就可以向服务器端发送异步请求并解析异步数据了，参考的写法如下。

【代码 11-2】

```
01 if (xhr != null) {
02     xhr.onreadystatechange = on_state_change;
03     xhr.open("GET|POST", url, true);
04     xhr.send(null);
05 } else {
06     // TODO: XMLHttpRequest Object is invalid.
07 }
```

关于【代码 11-2】的说明：

- 第 02 行代码通过 XMLHttpRequest 对象的 onreadystatechange 事件，定义了当异步请求发送到服务器端后需要执行的响应任务（相当于回调方法）。
- 第 03 行代码通过 XMLHttpRequest 对象的 open()方法定义异步请求的参数，GET|POST 规定了请求的类型，URL 定义了请求的链接地址，true 表示为异步处理请求。

- 第 04 行代码通过 XMLHttpRequest 对象的 send()方法执行将请求发送到服务器端的操作。另外，如果该方法带有 String 类型的参数（url 地址的参数集合），则代表请求类型必须为 POST 方式。

11.3　Ajax 解析文本

通过 Ajax 技术可以解析多种类型的数据和文件，其中最简单、最基础的就是解析文本文件。下面介绍一个通过 Ajax 实现解析文本操作的代码实例。

【代码 11-3】（详见源代码目录 ch11-js-ajax-txt.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript Ajax 应用 - 解析 txt 文本</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-ajax" style="">
    <input type="button" onclick="ajax_load_txt_request();"
            value="通过 Ajax 解析 txt 文本"/>
    <span>解析文本内容:</span>
    <span id="id-span-ajax-txt"></span>
</div>
</body>
<script type="text/javascript">
    var xhr = null;    // TODO: define XMLHttpRequest Object
    function ajax_load_txt_request() {
        if (window.XMLHttpRequest) {    // TODO: Chrome, Firefox, Opera,
                                           IE7+, Edge, etc.
            xhr = new XMLHttpRequest();
        } else if (window.ActiveXObject) {  // TODO: IE5, IE6
            xhr = new ActiveXObject("Microsoft.XMLHTTP");
        } else {
            xhr = null;
        }
```

```
29         if (xhr != null) {
30             xhr.onreadystatechange = on_state_change;
31             xhr.open("GET", "ajax.txt", true);
32             xhr.send(null);
33         } else {
34             console.log("Your browser does not support XMLHTTP.");
35         }
36     }
37     function on_state_change() {
38         if (xhr.readyState == 4) {
39             if (xhr.status == 200) {
40                 document.getElementById('id-span-ajax-txt').innerText =
                                                         xhr.responseText;
41             } else {
42                 console.log("Problem retrieving data:" + xhr.statusText);
43             }
44         }
45     }
46 </script>
47 </html>
```

关于【代码 11-3】的说明：

- 第 14 行代码通过<input>标签元素定义了一个按钮，并定义了单击 onclick 事件方法（ajax_load_txt_request()），用于执行 Ajax 异步解析文本文件的操作。
- 第 16 行代码通过<span>标签元素定义了一块行内区域，用于显示通过 Ajax 异步方式解析文本文件所获取的文本内容。
- 第 20 行代码定义了一个 XMLHttpRequest 对象实例（xhr），并初始化为 null。
- 第 21～36 行代码是 ajax_load_txt_request()方法的实现过程，具体内容如下：
  - 第 22～28 行代码用于创建 XMLHttpRequest 对象实例（见【代码 11-1】）。
  - 第 29～35 行代码用于执行 Ajax 异步请求操作（见【代码 11-2】）。其中，第 30 行代码定义了回调方法（on_state_change()），第 31 行代码调用的 open()方法定义了异步请求的文本文件路径，为本地的文本文件（"ajax.txt"）。
- 第 37～45 行代码是 on_state_change()回调方法的实现过程，具体内容如下：
  - 状态值 readyState 属性值为 4，表示响应内容解析完成。
  - 状态码 status 属性值为 200，表示异步操作成功。
  - XMLHttpRequest 对象的 responseText 属性值为解析后获取的文本文件（"ajax.txt"）中的内容。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 11.1 所示。

如图 11.1 中的箭头所示，右边页面中显示了通过 Ajax 异步方式解析文本文件（"ajax.txt"）所得到的内容。

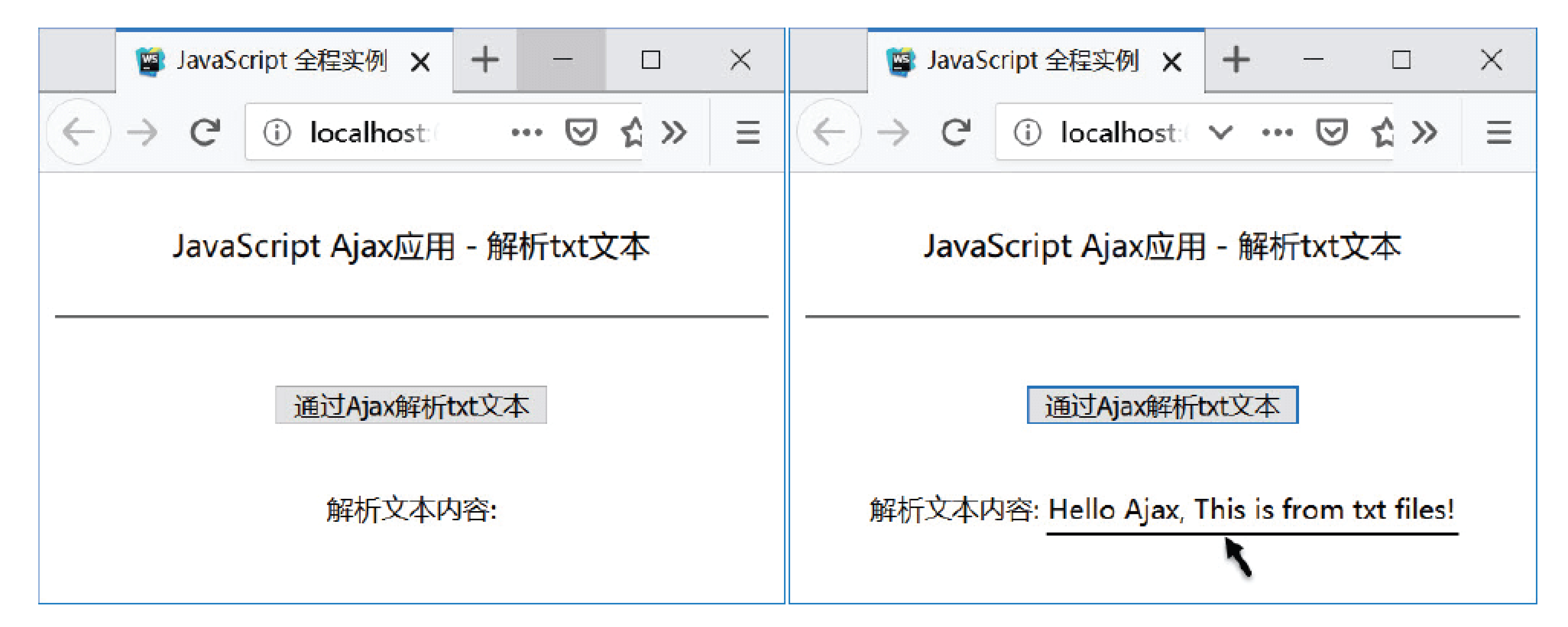


图 11.1　Ajax 实现解析文本

11.4　Ajax 解析 XML

上一节中介绍了通过 Ajax 技术解析文本文件的方法，本节继续介绍如何通过 Ajax 技术解析 XML 数据的方法。下面看一个 Ajax 解析 XML 文件并将获取的数据填充到表格中显示的代码实例。

【代码 11-4】（详见源代码目录 ch11-js-ajax-xml.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript Ajax 应用 - 解析 XML 数据</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-ajax" style="">
  <input type="button" onclick="ajax_load_xml_request();"
         value="通过 Ajax 解析 XML 数据"/>
    <div id="id-div-ajax-txt"></div>
    <div id="id-div-ajax-xml"></div>
</div>
</body>
<script type="text/javascript">
    var xhr = null; // TODO: define XMLHttpRequest Object
    function ajax_load_xml_request() {
```

```
        if (window.XMLHttpRequest) {    // TODO: Chrome, Firefox, Opera, IE7+, Edge, etc.
            xhr = new XMLHttpRequest();
        } else if (window.ActiveXObject) {  // TODO: IE5, IE6
            xhr = new ActiveXObject("Microsoft.XMLHTTP");
        } else {
            xhr = null;
        }
        if (xhr != null) {
            xhr.onreadystatechange = on_state_change;
            xhr.open("GET", "ajax.xml", true);
            xhr.send(null);
        } else {
            console.log("Your browser does not support XMLHTTP.");
        }
    }
    function on_state_change() {
        if (xhr.readyState == 4) {
            if (xhr.status == 200) {
                document.getElementById('id-div-ajax-txt').innerText = xhr.responseText;
                var xmlDoc = xhr.responseXML;
                var x = xmlDoc.documentElement.getElementsByTagName("WEB");
        var v = "<table border='1'><tr><th>Title</th><th>Address</th><th>Info</th></tr>";
                var len = x.length;
                var xx = "";
                for (let i = 0; i < len; i++) {
                    v += "<tr>";
                    xx = x[i].getElementsByTagName('TITLE');
                    {
                        try {
                            v += "<td>" + xx[0].firstChild.nodeValue + "</td>";
                        } catch (er) {
                            v += "<td> </td>";
                        }
                    }
                    xx = x[i].getElementsByTagName('ADDRESS');
                    {
                        try {
                            v += "<td>" + xx[0].firstChild.nodeValue + "</td>";
                        } catch (er) {
                            v += "<td> </td>";
```

```
62                          }
63                      }
64                      xx = x[i].getElementsByTagName('INFO');
65                      {
66                          try {
67                              v += "<td>" + xx[0].firstChild.nodeValue + "</td>";
68                          } catch (er) {
69                              v += "<td> </td>";
70                          }
71                      }
72                      v += "</tr>";
73                  }
74                  v += "</table>";
75                  document.getElementById('id-div-ajax-xml').innerHTML = v;
76              } else {
77                  console.log("Problem retrieving data:" + xhr.statusText);
78              }
79          }
80      }
81  </script>
82  </html>
```

关于【代码 11-4】的说明：

- 第 14 行代码通过<input>标签元素定义了一个按钮，并定义了单击 onclick 事件方法（ajax_load_xml_request()），用于执行 Ajax 异步解析 XML 文件的操作。
- 第 15 行代码通过<div id="id-div-ajax-txt">标签元素定义了一个层区域，用于显示通过 Ajax 异步方式解析 XML 文件所获取的文本内容。
- 第 16 行代码通过<div id="id-div-ajax-xml">标签元素定义了另一个层区域，用于显示通过 Ajax 异步方式解析 XML 文件后将所获取的数据填充到一个动态生成的表格。
- 第 20 行代码定义了一个 XMLHttpRequest 对象实例（xhr），并初始化为 null。
- 第 21 ~ 36 行代码是 ajax_load_xml_request()方法的实现过程，具体内容如下：
  - 第 22 ~ 28 行代码用于创建 XMLHttpRequest 对象实例（见【代码 11-1】）。
  - 第 29 ~ 35 行代码用于执行 Ajax 异步请求操作（见【代码 11-2】）。其中，第 30 行代码定义了回调方法（on_state_change()），第 31 行代码调用的 open()方法定义了异步请求的 XML 文件路径，为本地的 XML 文件（"ajax.xml"）。
- 第 37 ~ 80 行代码是 on_state_change()回调方法的实现过程（参考【代码 11-3】），需要介绍的内容如下：
  - 第 40 行代码通过 XMLHttpRequest 对象的 responseText 属性获取了解析 XML 文件（"ajax.xml"）中的内容，并通过层（<div id="id-div-ajax-txt">）元素进行显示。

➢ 第 41 行代码通过 XMLHttpRequest 对象的 responseXML 属性将解析 XML 文件（"ajax.xml"）的结果作为 Document 对象返回。
➢ 第 42～75 行代码将通过 Ajax 异步方式获取的 XML 文件（"ajax.xml"）数据填充进一个动态生成的表格中，并通过层（<div id="id-div-ajax-xml">）元素进行显示。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 11.2 和图 11.3 所示。

如图 11.2 所示，页面中显示了通过 Ajax 异步方式解析 XML 文件（"ajax.xml"）所得到的文本内容。

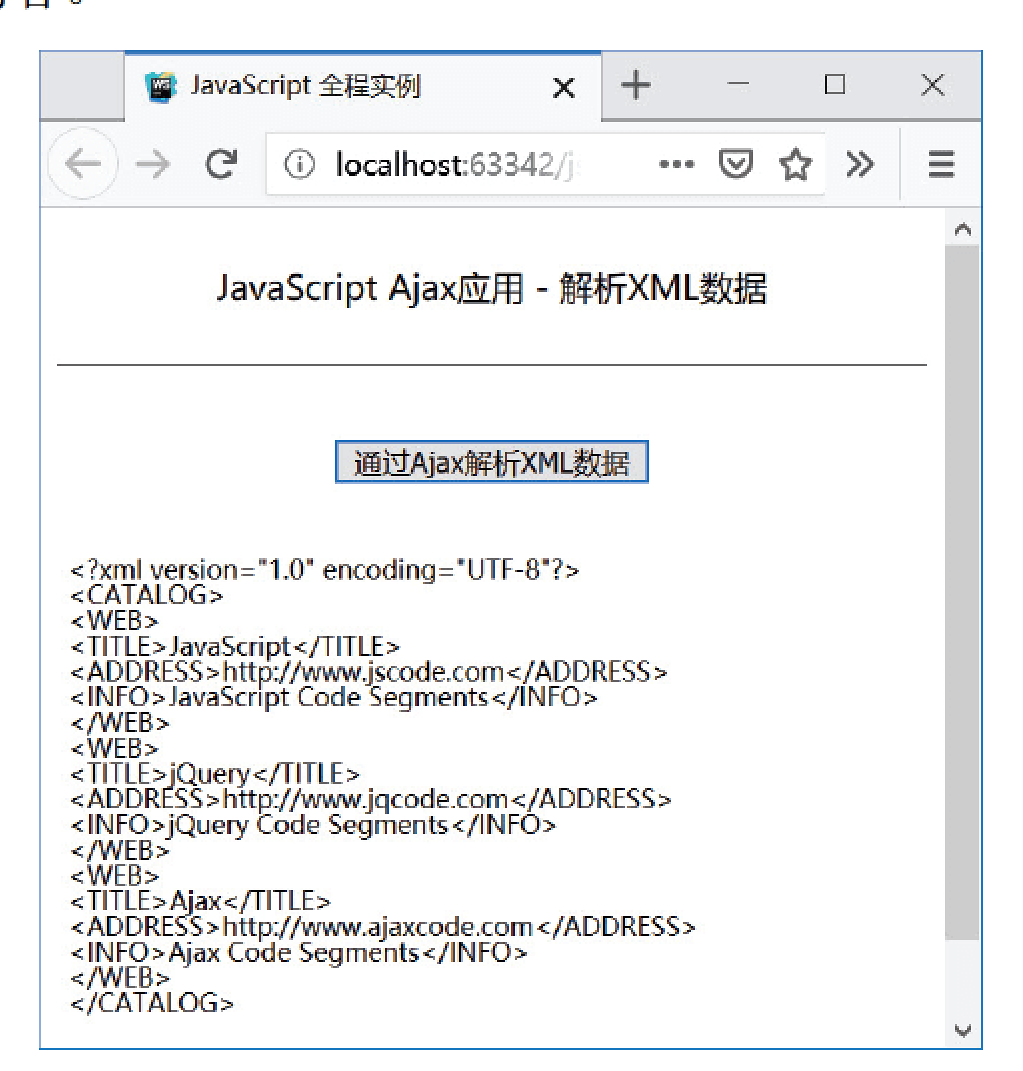


图 11.2　Ajax 实现解析 XML 文件（文本方式）

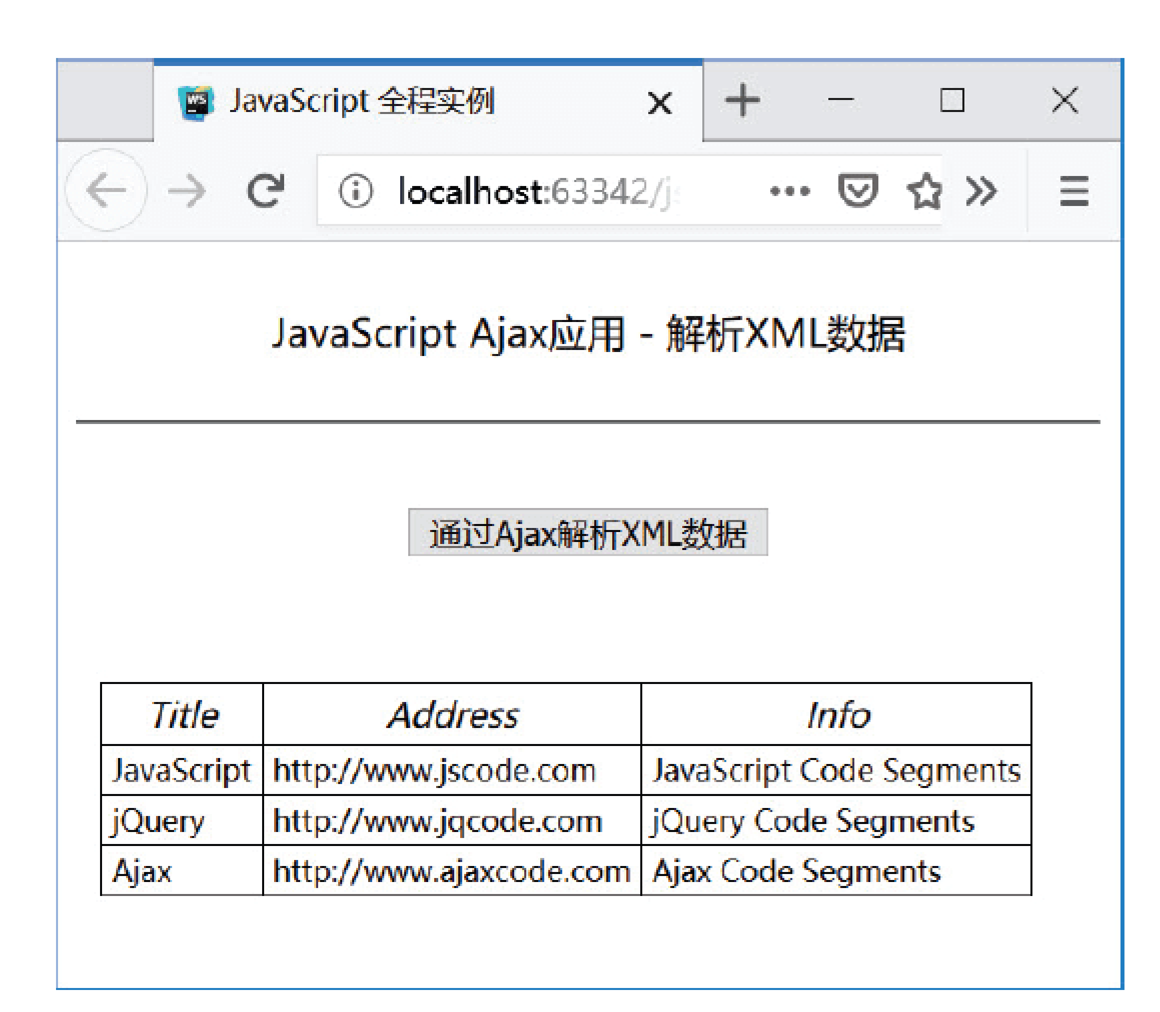


图 11.3　Ajax 实现解析 XML 文件（XML 方式）

如图 11.3 所示，页面中显示了通过 Ajax 异步方式解析 XML 文件（"ajax.xml"）并将返回的数据填充进一个动态生成的表格中的效果。由此可见，Ajax 应用使用严格的 XML 文件的效果会更好，数据操作也更快速、简洁。

11.5　Ajax 解析 JSON

在 8.4 节中介绍了通过 Ajax 解析文本文件和 XML 文件的方法，本节将介绍如何通过 Ajax 技术解析 JSON 格式数据的方法。所谓 JSON（JavaScript Object Notation），是指一种 JavaScript 对象标记，是遵循 ECMAScript 规范标准设计的，是一种目前非常流行的轻量级数据交换格式。下面看一个 Ajax 解析 JSON 文件并将获取的数据填充到表格中显示的代码实例。

【代码 11-5】（详见源代码目录 ch11-js-ajax-json.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
```

```
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript Ajax 应用 - 解析 JSON 数据</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-ajax" style="">
<input type="button" onclick="ajax_load_json_request();"
        value="通过 Ajax 解析 JSON 数据"/>
    <div id="id-div-ajax-txt"></div>
    <div id="id-div-ajax-json"></div>
</div>
</body>
<script type="text/javascript">
    var xhr = null; // TODO: define XMLHttpRequest Object
    function ajax_load_json_request() {
        if (window.XMLHttpRequest) {    // TODO: Chrome, Firefox, Opera,
                                           IE7+, Edge, etc.
            xhr = new XMLHttpRequest();
        } else if (window.ActiveXObject) {  // TODO: IE5, IE6
            xhr = new ActiveXObject("Microsoft.XMLHTTP");
        } else {
            xhr = null;
        }
        if (xhr != null) {
            xhr.onreadystatechange = on_state_change;
            xhr.open("GET", "ajax.json", true);
            xhr.send(null);
        } else {
            console.log("Your browser does not support XMLHTTP.");
        }
    }
    function on_state_change() {
        if (xhr.readyState == 4) {
            if (xhr.status == 200) {
                var jsonDoc = xhr.responseText;
                document.getElementById('id-div-ajax-txt').
                                                   innerText = jsonDoc;
                var jsonObj = JSON.parse(jsonDoc);
                var jsonWeb = jsonObj.web;
                var len = jsonWeb.length;
```

```
45          var v = "<table border='1'><tr><th>Title</th><th>Address</th><th>Info</th></tr>";
46                  for (let i = 0; i < len; i++) {
47                      v += "<tr>";
48                      {
49                          try {
50                              v += "<td>" + jsonWeb[i].title + "</td>";
51                          } catch (err) {
52                              v += "<td> </td>";
53                          }
54                      }
55                      {
56                          try {
57                              v += "<td>" + jsonWeb[i].address + "</td>";
58                          } catch (err) {
59                              v += "<td> </td>";
60                          }
61                      }
62                      {
63                          try {
64                              v += "<td>" + jsonWeb[i].info + "</td>";
65                          } catch (err) {
66                              v += "<td> </td>";
67                          }
68                      }
69                      v += "</tr>";
70                  }
71                  v += "</table>";
72                  document.getElementById('id-div-ajax-json').innerHTML = v;
73              } else {
74                  console.log("Problem retrieving data:" + xhr.statusText);
75              }
76          }
77      }
78  </script>
79  </html>
```

关于【代码 11-5】的说明：

- 第 14 行代码通过<input>标签元素定义了一个按钮，并定义了单击 onclick 事件方法（ajax_load_json_request()），用于执行 Ajax 异步解析 JSON 文件的操作。
- 第 15 行代码通过<div id="id-div-ajax-txt">标签元素定义了一个层区域，用于显示通过 Ajax 异步方式解析 JSON 文件所获取的文本内容。

- 第 16 行代码通过<div id="id-div-ajax-json">标签元素定义了另一个层区域，用于显示通过 Ajax 异步方式解析 JSON 文件后将所获取的数据填充到一个动态生成的表格中。
- 第 20 行代码定义了一个 XMLHttpRequest 对象实例（xhr），并初始化为 null。
- 第 21～36 行代码是 ajax_load_json_request()方法的实现过程，具体内容如下：
  - 第 22～28 行代码用于创建 XMLHttpRequest 对象实例（见【代码 11-1】）。
  - 第 29～35 行代码用于执行 Ajax 异步请求操作（见【代码 11-2】）。其中，第 30 行代码定义了回调方法（on_state_change()），第 31 行代码调用的 open()方法定义了异步请求的 JSON 文件路径，为本地的文本文件（"ajax.json"）。
- 第 37～45 行代码是 on_state_change()回调方法的实现过程（参考【代码 11-3】），需要介绍的内容如下：
  - 第 40 行代码通过 XMLHttpRequest 对象的 responseText 属性获取了解析 JSON 文件（"ajax.json"）中的内容，并以文本形式保存在变量 jsonDoc 中，然后在第 41 行代码中通过层（<div id="id-div-ajax-txt">）元素进行显示。
  - 第 42 行代码通过 JSON 对象的 parse()方法（将文本转化为 JSON 对象格式）将变量 jsonDoc 解析为 JSON 对象（变量 jsonObj）。
  - 第 43 行代码通过变量 jsonObj 获取了 JSON 文件（"ajax.json"）中定义的 web 数组，并保存在变量 jsonWeb 中。这里需要强调的是，JSON 对象数据格式就是直接通过键名来获取键值的（区别于 XML 格式）。
  - 第 45～72 行代码通过变量 jsonWeb 直接操作键名获取键值，然后填充进一个动态生成的表格中，并通过层（<div id="id-div-ajax-json">）元素进行显示。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 11.4 和图 11.5 所示。

如图 11.4 所示，页面中显示了通过 Ajax 异步方式解析 JSON 文件（"ajax.json"）所得到的文本内容，注意 JSON 格式与 XML 格式的区别（更加简洁了）。

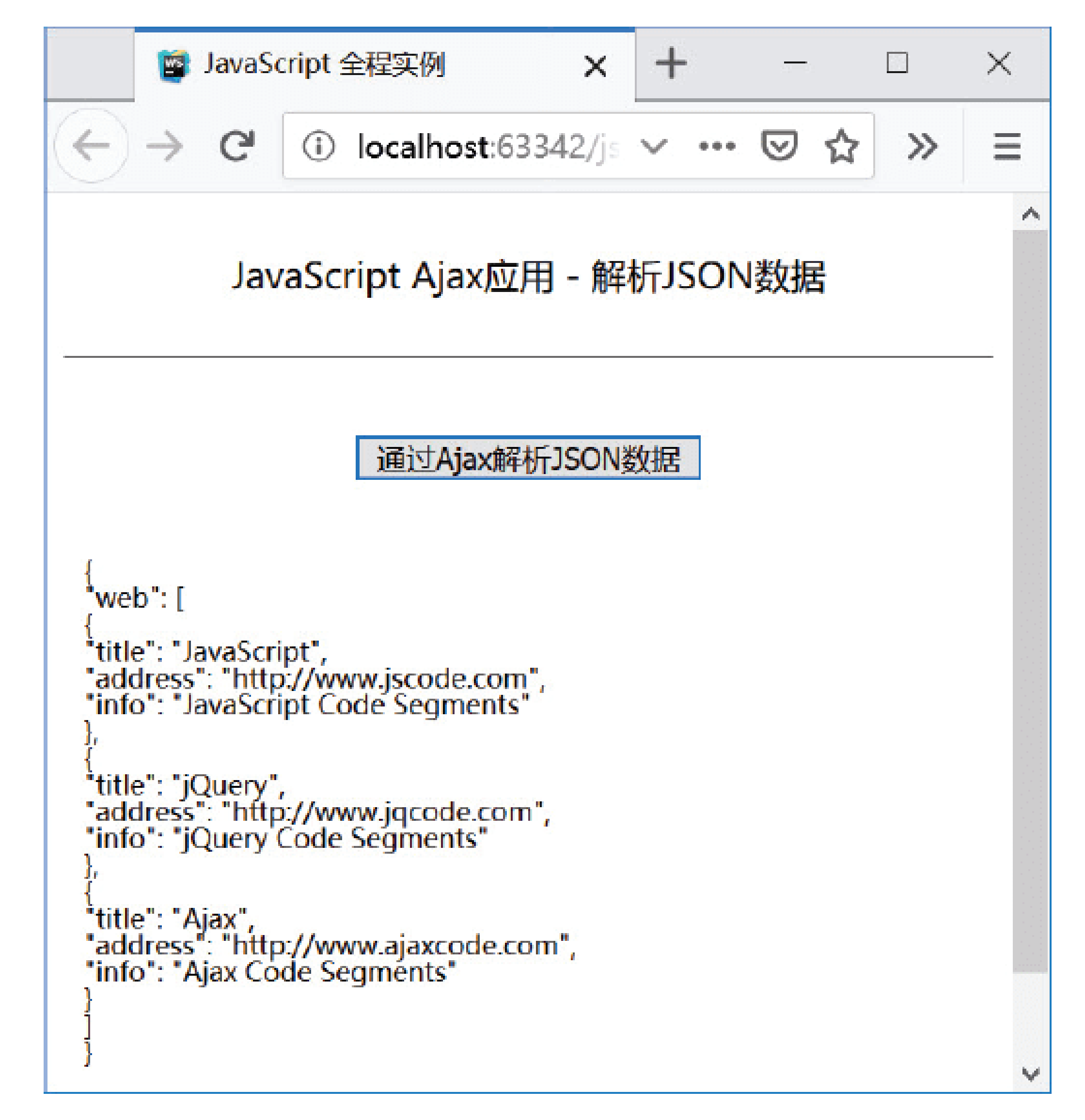
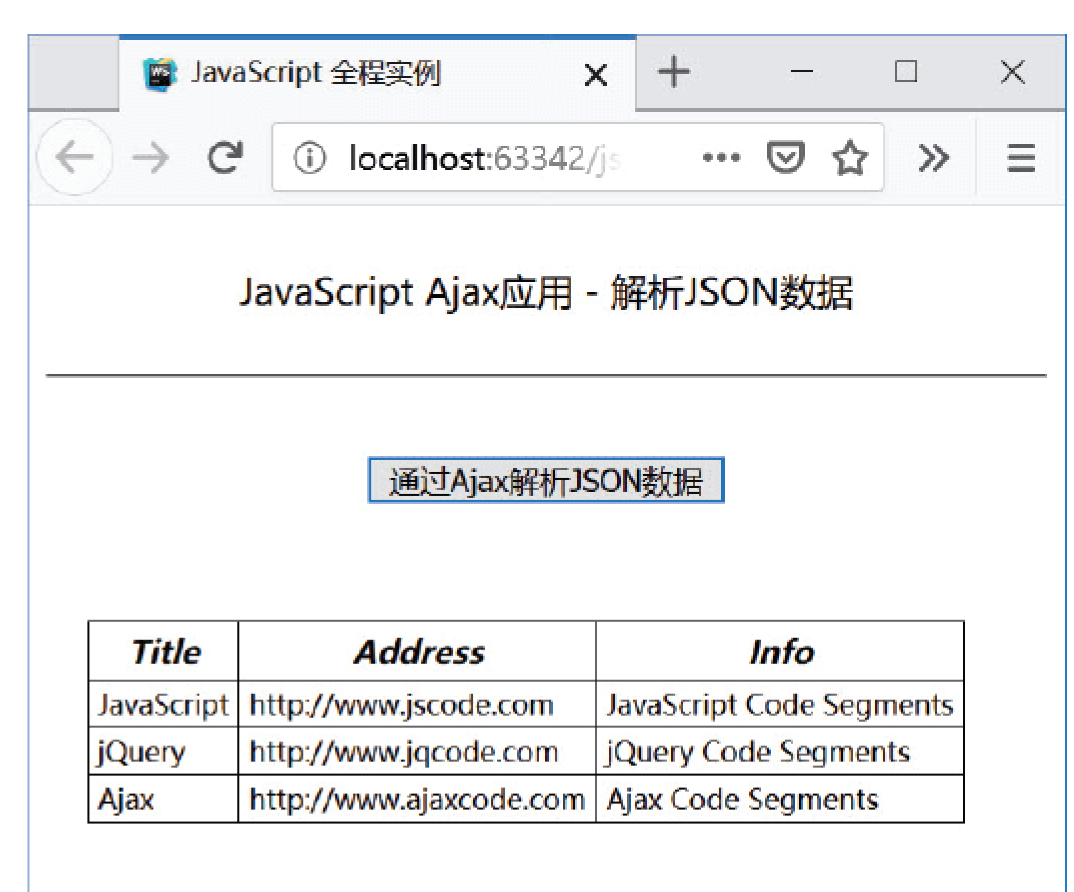



图 11.4　Ajax 实现解析 JSON 文件（文本方式）



Title	Address	Info
JavaScript	http://www.jscode.com	JavaScript Code Segments
jQuery	http://www.jqcode.com	jQuery Code Segments
Ajax	http://www.ajaxcode.com	Ajax Code Segments

图 11.5　Ajax 实现解析 JSON 文件（JSON 方式）

如图 11.5 所示，页面中显示了通过 Ajax 异步方式解析 JSON 文件（"ajax.json"）并将返回的数据填充进一个动态生成的表格中的效果。

11.6 实现一个 Ajax 框架

前文中陆续介绍了通过 Ajax 解析文本文件、XML 文件和 JSON 格式文件的方法，细心的读者一定会发现创建 Ajax 应用的代码基本都有固定的流程和模式。设计人员可以通过编写 Ajax 框架来构建模块化的开发流程，进而提高代码的使用效率。目前，主流的 JavaScript 框架（如具有代表性的 jQuery 框架）均实现了 Ajax 模块化功能，以便于设计人员使用 Ajax 功能。

在本节中，我们就自己设计一个非常简单的 Ajax 框架，让读者大致了解一下开发框架的基本流程。下面看一下具体的 JavaScript 代码实例。

【代码 11-6】（详见源代码目录 ch11-js-ajax-myframework.html 文件）

```
<script type="text/javascript">
    /**
     * my ajax framework
     */
    function myAjaxFramework(options) {
        var resType;    // TODO: define file type
        if (!options || !options.url) { // TODO: 检测是否存在请求的 URL
            return false;
        }
        if (/txt/.test(options.url)) {
            resType = "text";
        } else if (/xml/.test(options.url)) {
            resType = "xml";
        } else if (/json/.test(options.url)) {
            resType = "json";
        } else {
            resType = "";
        }
        /**
         * 数据初始化
         */
        options.data = options.data || "";     // TODO: 待传送的值
        options.method = (options.method || "GET").toUpperCase();
                                          // TODO:默认是 GET
        options.async = options.async || true;
                                          // TODO: 异步（true）或同步（false）
```

```
        options.responseType = options.responseType || resType;
                                                // TODO: response type
        options.successCall = options.successCall || false;
                                                // TODO: 成功回调
        options.failureCall = options.failureCall || false;
                                               // TODO: 失败回调
        // TODO: define XMLHttpRequest
        var xhr;
        // TODO: create XMLHttpRequest
        if (window.XMLHttpRequest) {    // TODO: Chrome, Firefox, Opera,
                                           IE7+, Edge, etc.
            xhr = new XMLHttpRequest();
        } else if (window.ActiveXObject) {         // TODO: IE5, IE6
            xhr = new ActiveXObject("Microsoft.XMLHTTP");
        } else {
            xhr = null;
        }
        if (xhr != null) {
            xhr.onreadystatechange = function () {
                if (xhr.readyState == 4) {
                    if (xhr.status == 200) {
                        if (options.successCall) {
                    options.successCall(getResponseData(xhr,
                                                options.responseType));
                        }
                    } else {
                        if (options.failureCall) {
                            options.failureCall(xhr, xhr.status);
                        }
                    }
                }
            };
            xhr.open(options.method,options.url+(options.method==
                                  "GET"? "?"+options.data:""),options.async);
            if (options.method != "GET" && options.data) {
                xhr.send(options.data);
            } else {
                xhr.send();
            }
        } else {
            console.log("Your browser does not support XMLHTTP.");
        }
        return true; // TODO: myAjaxFramework 调用成功
```

```
62     }
63     /**
64      * get response data
65      */
66     function getResponseData(xhr, resType) {
67         if (resType === "text") {
68             return {resType: resType, resData: xhr.responseText};
69         } else if (resType === "xml") {
70             return {resType: resType, resData: xhr.responseXML};
71         } else if (resType === "json") {
72             return {resType: resType, resData:
                                              JSON.parse(xhr.responseText)};
73         } else {
74             return {};
75         }
76     }
77 </script>
```

关于【代码 11-6】的说明：

- 第 05～62 行代码是我们定义的 Ajax 框架（myAjaxFramework）的实现过程，具体内容如下：
  - 第 05 行代码以函数方式（function）定义了 Ajax 框架（myAjaxFramework），同时定义了一个参数（options），用来传递 Ajax 相关属性。
  - 第 06 行代码定义了一个变量（resType），用来保存异步加载文件的类型（文本 TXT、XML 或 JSON）。
  - 第 10～18 行代码借助正则表达式的方式来获取异步加载文件的类型（TEXT、XML 或 JSON），并保存在变量（resType）中。
  - 第 22～27 行代码用于初始化参数（options），包括异步传输的数据（data）、请求方式（GET 或 POST）、文件类型（resType）、成功回调方法（successCall）和失败回调方法（failureCall）等。
  - 第 29～37 行代码定义并创建了 XMLHttpRequest 对象实例（xhr）。
  - 第 38～60 行代码用于执行 Ajax 异步请求操作。关键代码包括：第 43 行代码调用了成功回调方法（successCall），并通过自定义方法（getResponseData()）获取了异步返回的数据；第 52 行代码调用的 open()方法打开异步请求的文件；第 53～57 行代码用于判断请求类型（GET 或 POST），并调用 send()方法执行异步请求的操作。
- 第 66～76 行代码是自定义 getResponseData()方法的实现过程，主要是根据异步请求文件的类型来获取异步请求的返回数据（responseText 或.responseXML）。

以上就是我们实现的一个非常简单的 Ajax 框架（myAjaxFramework）的过程，这个框架主要用来自动区分文件类型并实现异步加载功能。

11.7 使用 Ajax 框架轻松加载文件

在本节中，我们将在刚刚完成的 Ajax 框架（myAjaxFramework）的基础上实现轻松异步加载文件的操作。下面看一下具体的 JavaScript 代码实例。

【代码 11-7】（详见源代码目录 ch11-js-ajax-load-files.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript Ajax 应用 - 使用 Ajax 轻松加载文件</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-ajax" style="">
    <input type="button" onclick="on_my_ajax();"
                         value="通过 Ajax 框架加载文件"/>
    <div>解析文件内容:</div>
    <div id="id-div-ajax-response"></div>
</div>
</body>
<script type="text/javascript">
    /**
     * my ajax framework
     */
    function myAjaxFramework(options) {
        var resType;    // TODO: define file type
        if (!options || !options.url) { // TODO: 检测是否存在请求的 URL
            return false;
        }
        if (/txt/.test(options.url)) {
            resType = "text";
        } else if (/xml/.test(options.url)) {
            resType = "xml";
        } else if (/json/.test(options.url)) {
```

```
            resType = "json";
        } else {
            resType = "";
        }
        /**
         * 数据初始化
         */
        options.data = options.data || "";    // TODO: 待传送的值
        options.method = (options.method || "GET").toUpperCase();
                                              // TODO:默认是 GET
        options.async = options.async || true;
                                              // TODO: 异步（true）或同步（false）
        options.responseType = options.responseType || resType;
                                              // TODO:response type
        options.successCall = options.successCall || false;
                                              // TODO: 成功回调
        options.failureCall = options.failureCall || false;
                                              // TODO: 失败回调
        // TODO: define XMLHttpRequest
        var xhr;
        // TODO: create XMLHttpRequest
        if (window.XMLHttpRequest) {      // TODO: Chrome, Firefox, Opera,
                                                IE7+, Edge, etc.
            xhr = new XMLHttpRequest();
        } else if (window.ActiveXObject) {  // TODO: IE5, IE6
            xhr = new ActiveXObject("Microsoft.XMLHTTP");
        } else {
            xhr = null;
        }
        if (xhr != null) {
            xhr.onreadystatechange = function () {
                if (xhr.readyState == 4) {
                    if (xhr.status == 200) {
                        if (options.successCall) {
        options.successCall(getResponseData(xhr, options.url,
                                             options.responseType));
                        }
                    } else {
                        if (options.failureCall) {
                            options.failureCall(xhr, xhr.status);
                        }
```

```
                    }
                }
            };
    xhr.open(options.method, options.url + (options.method == "GET" ? "?"
            + options.data : ""), options.async);
            if (options.method != "GET" && options.data) {
                xhr.send(options.data);
            } else {
                xhr.send();
            }
        } else {
            console.log("Your browser does not support XMLHTTP.");
        }

        return true; // TODO: myAjaxFramework 调用成功
    }
    /**
     * get response data
     */
    function getResponseData(xhr, resUrl, resType) {
        if (resType === "text") {
            return {resType: resType, resUrl: resUrl, resData:
                    xhr.responseText};
        } else if (resType === "xml") {
            return {resType: resType, resUrl: resUrl, resData:
                    xhr.responseXML};
        } else if (resType === "json") {
        return {resType: resType, resUrl: resUrl, resData:
                    JSON.parse(xhr.responseText)};
        } else {
            return {};
        }
    }
    /**
     * on_my_ajax
     */
    function on_my_ajax() {
        /**
         * call myAjaxFramework
         */
        myAjaxFramework({
```

```
            // TODO: 定义 url
            "url": "ajax.txt",
            // TODO: 成功回调
            "successCall": function (resObj) {
                loadFiles(resObj);
            },
            // TODO: 失败回调
            "failureCall": function (xmlRes, resCode) {
                console.log(resCode);
            }
        });
        /**
         * call myAjaxFramework
         */
        myAjaxFramework({
            // TODO: 定义 url
            "url": "ajax.xml",
            // TODO: 成功回调
            "successCall": function (resObj) {
                loadFiles(resObj);
            },
            // TODO: 失败回调
            "failureCall": function (xmlRes, resCode) {
                console.log(resCode);
            }
        });
        /**
         * call myAjaxFramework
         */
        myAjaxFramework({
            // TODO: 定义 url
            "url": "ajax.json",
            // TODO: 成功回调
            "successCall": function (resObj) {
                loadFiles(resObj);
            },
            // TODO: 失败回调
            "failureCall": function (xmlRes, resCode) {
                console.log(resCode);
            }
        });
    }
```

```
    /**
     * load files
     * @param resObj
     */
    function loadFiles(resObj) {
        var vUrl, vData;
        switch (resObj.resType) {
            case "text":
                vUrl = "异步加载文本文件: " + resObj.resUrl + "<br>";
                vData = resObj.resData + "<br><br>";
            document.getElementById("id-div-ajax-response").
                                        innerHTML += vUrl + vData;
                break;
            case "xml":
                vUrl = "异步加载 XML 文件: " + resObj.resUrl + "<br>";
            var x = resObj.resData.documentElement.
                                            getElementsByTagName("WEB");
    var vData = "<table border='1'><tr><th>Title</th><th>Address</th>
                    <th>Info</th></tr>";
                var len = x.length;
                var xx = "";
                for (let i = 0; i < len; i++) {
                    vData += "<tr>";
                    xx = x[i].getElementsByTagName('TITLE');
                    {
                        try {
                            vData += "<td>" + xx[0].firstChild.nodeValue +
                                    "</td>";
                        } catch (er) {
                            vData += "<td> </td>";
                        }
                    }
                    xx = x[i].getElementsByTagName('ADDRESS');
                    {
                        try {
                            vData += "<td>" + xx[0].firstChild.nodeValue +
                                    "</td>";
                        } catch (er) {
                            vData += "<td> </td>";
                        }
                    }
                    xx = x[i].getElementsByTagName('INFO');
```

```
                    {
                        try {
                            vData += "<td>" + xx[0].firstChild.nodeValue +
                                       "</td>";
                        } catch (er) {
                            vData += "<td> </td>";
                        }
                    }
                    vData += "</tr>";
                }
                vData += "</table>";
                vData += "<br><br>";
            document.getElementById("id-div-ajax-response").innerHTML +=
                                                        vUrl + vData;
                break;
            case "json":
                vUrl = "异步加载 JSON 文件: " + resObj.resUrl + "<br>";
                var jsonObj = resObj.resData;
                //var jsonStringify = JSON.stringify(jsonDoc);
                var jsonWeb = jsonObj.web;
                var len = jsonWeb.length;
    var vData = "<table border='1'><tr><th>Title</th><th>Address</th>
                    <th>Info</th></tr>";
                for (let i = 0; i < len; i++) {
                    vData += "<tr>";
                    {
                        try {
                            vData += "<td>" + jsonWeb[i].title + "</td>";
                        } catch (err) {
                            vData += "<td> </td>";
                        }
                    }
                    {
                        try {
                            vData += "<td>" + jsonWeb[i].address + "</td>";
                        } catch (err) {
                            vData += "<td> </td>";
                        }
                    }
                    {
                        try {
                            vData += "<td>" + jsonWeb[i].info + "</td>";
                        } catch (err) {
```

```
                            vData += "<td> </td>";
                        }
                    }
                    vData += "</tr>";
                }
                vData += "</table>";
                vData += "<br><br>";
            document.getElementById("id-div-ajax-response").innerHTML +=
                                         vUrl + vData;
                break;
            default:
                break;
        }
    }
</script>
</html>
```

关于【代码 11-7】的说明：

- 【代码 11-7】是在【代码 11-6】(myAjaxFramework 框架)、【代码 11-3】(异步加载文本文件)、【代码 11-4】(异步加载 XML 文件)和【代码 11-5】(异步加载 JSON 文件)的基础上改进而成的。
- 自定义方法 getResponseData()略做了修改，增加了一个文件路径参数，可以获取异步加载文件的路径。
- 自定义方法 loadFiles()整合了【代码 11-3】(异步加载文本文件)、【代码 11-4】(异步加载 XML 文件)和【代码 11-5】(异步加载 JSON 文件)的代码，实现了将异步加载的 XML 文件和 JSON 文件内容转化为表格形式的功能。

下面使用 Firefox 浏览器运行测试该 HTML 网页，初始效果如图 11.6 所示。



图 11.6　Ajax 框架轻松加载文件（初始状态）

如图 11.6 中的箭头所示，在页面中单击“通过 Ajax 框架加载文件”按钮，文件异步加载的效果如图 11.7 所示。

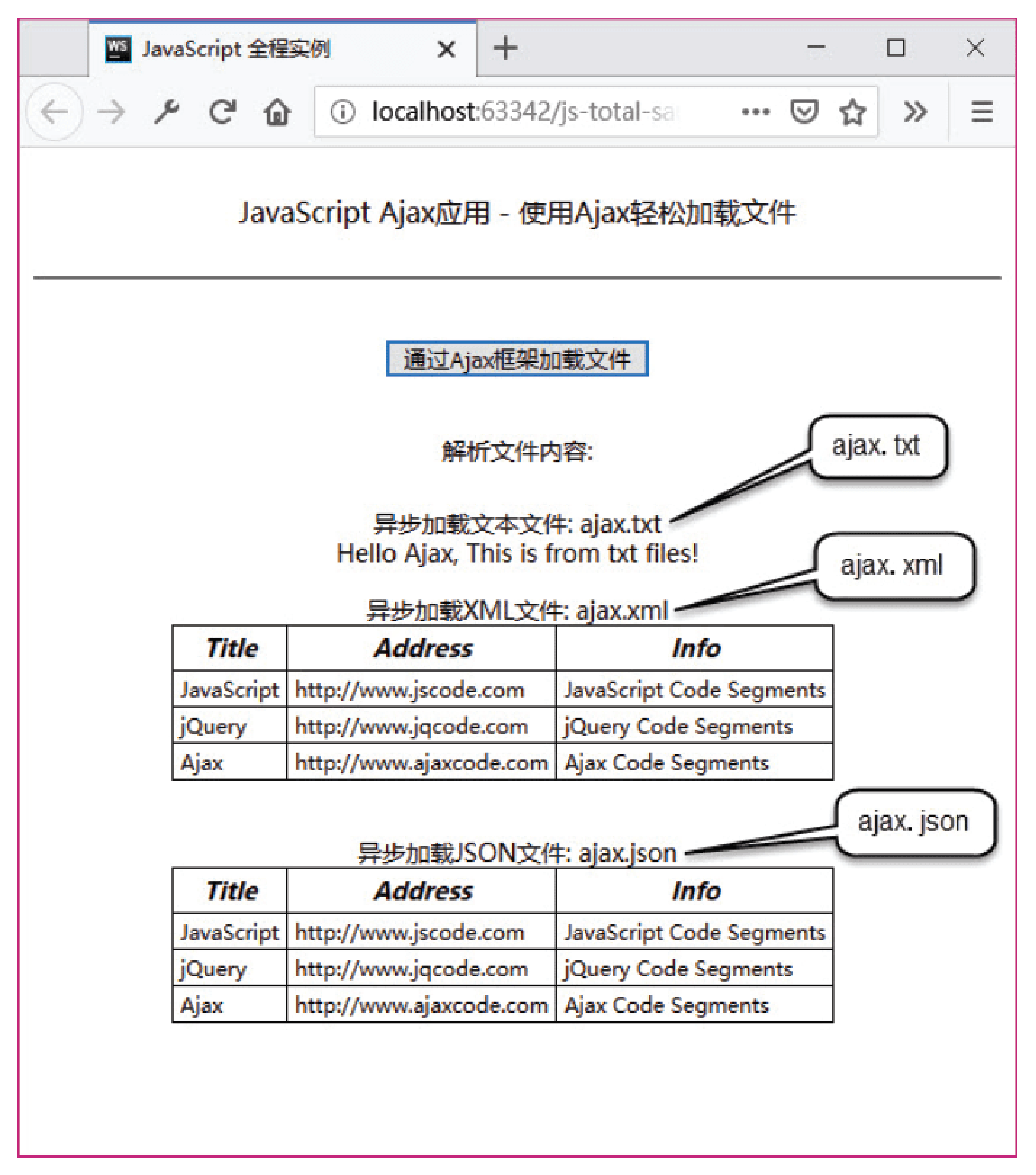


图 11.7　Ajax 框架轻松加载文件（加载完成）

如图 11.7 中的标识所示，页面中显示了通过 Ajax 异步方式加载文本文件（ajax.txt）、XML 文件（ajax.xml）和 JSON 文件（"ajax.json"）的效果。

11.8　Ajax 跨域异步交互

Ajax 技术真正发挥作用的地方是能够实现跨域与服务器进行异步数据交互功能。Ajax 技术的优势还体现在数据加载过程中不会重新加载整个页面、仅仅是局部刷新网页内容，显著增强了网页浏览的用户体验效果。下面看一个通过 Ajax 实现跨域异步交互（PHP 服务器）的 JavaScript 代码实例。

【代码 11-8】（详见源代码目录 ch11-js-ajax-php.html 文件）

```
<!doctype html>
<html lang="en">
<head>
    <!-- 添加文档头部内容 -->
    <title>JavaScript 全程实例</title>
</head>
```

```
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript Ajax 应用 - Ajax 跨域异步交互</nav>
</header>
<!-- 添加文档主体内容 -->
<div id="id-div-ajax" style="">
    <input type="button" onclick="ajax_load_php_request();"
                                value="通过 Ajax 解析 PHP"/>
    <div>解析 PHP 文件:</div>
    <div id="id-div-ajax-php"></div>
</div>
</body>
<script type="text/javascript">
    var xhr = null; // TODO: define XMLHttpRequest Object
    /**
     * request php file by Ajax
     */
    function ajax_load_php_request() {
        if (window.XMLHttpRequest) {    // TODO: Chrome, Firefox, Opera,
                                                 IE7+, Edge, etc.
            xhr = new XMLHttpRequest();
        } else if (window.ActiveXObject) {  // TODO: IE5, IE6
            xhr = new ActiveXObject("Microsoft.XMLHTTP");
        } else {
            xhr = null;
        }
        if (xhr != null) {
            xhr.onreadystatechange = on_state_change;
            xhr.open("GET", "ajax.php", true);
            xhr.send(null);
        } else {
            console.log("Your browser does not support XMLHTTP.");
        }
    }
    /**
     * callback func
     */
    function on_state_change() {
        var resText, iPos, vData, resJson, jsonObj, iLen;
        if (xhr.readyState == 4) {
            if (xhr.status == 200) {
```

```
            resText = xhr.responseText;
            iPos = resText.indexOf('!');
            vData = resText.substr(0, iPos + 1);
            resJson = resText.substr(iPos + 1);
            if(resJson.indexOf('{') > -1) {
                jsonObj = JSON.parse(resJson);
                iLen = jsonObj.length;
  vData += "<table border='1'><tr><th>Title</th><th>Address</th>
                                                      <th>Info</th></tr>";
                for (let i = 0; i < iLen; i++) {
                    vData += "<tr>";
                    {
                        try {
                            vData += "<td>" + jsonObj[i].title + "</td>";
                        } catch (err) {
                            vData += "<td> </td>";
                        }
                    }
                    {
                        try {
                            vData += "<td>" + jsonObj[i].address + "</td>";
                        } catch (err) {
                            vData += "<td> </td>";
                        }
                    }
                    {
                        try {
                            vData += "<td>" + jsonObj[i].info + "</td>";
                        } catch (err) {
                            vData += "<td> </td>";
                        }
                    }
                    vData += "</tr>";
                }
                vData += "</table>";
                document.getElementById('id-div-ajax-php').
                                                           innerHTML = vData;
            } else {
                document.getElementById('id-div-ajax-php').
                                                           innerText = resText;
            }
        } else {
```

```
86                  console.log("Problem retrieving data:" + xhr.statusText);
87              }
88          }
89      }
90  </script>
91  </html>
```

关于【代码 11-8】的说明：

- 【代码 11-8】与前面的代码实例不同，异步交互的是 PHP 服务器端的文件（第 34 行代码定义的 ajax.php 文件）。服务器端文件的运行与本地文件不同，网页需要在搭建的服务器上运行才可以得到正确的结果。

下面将 PHP 文件的代码简单介绍一下。

【代码 11-9】（详见源代码目录 ajax.php 文件）

```
01  <?php
02  echo "Hello, this is from server php file by ajax!";
03  $webArr = array(
04      array("title"=>"JavaScript",
05              "address"=>"http://www.jscode.com",
06              "info"=>"JavaScript Code Segments"),
07      array("title"=>"jQuery",
08              "address"=>"http://www.jqcode.com",
09              "info"=>"jQuery Code Segments"),
10      array("title"=>"Ajax",
11              "address"=>"http://www.ajax.com",
12              "info"=>"Ajax Code Segments")
13  );
14  echo json_encode($webArr);  //将数组进行 json 编码
15  ?>
```

关于【代码 11-9】的说明：

- 第 02 行代码通过 echo 语句回写一行文本。
- 第 03 ~ 13 行代码定义了一个二维数组（webArr），第 14 行代码通过 json_encode()语句将二维数组（webArr）进行 JSON 格式编码，并通过 echo 语句回写。
- 这样，PHP 文件就同时将一行字符串文本和一个 JSON 对象回写，交由 Ajax 进行异步处理。

下面使用 Firefox 浏览器运行测试该 HTML 网页，初始效果如图 11.8 所示。在无服务器的环境下会直接将 PHP 文件（ajax.php）解析为文本进行显示。读者可以注意到，解析得到的文本是全部 PHP 代码。

在 PHP 服务器的环境下运行，页面的效果如图 11.9 所示，得到了解析的字符串文本以及将 JSON 对象转化为表格形式显示的效果。

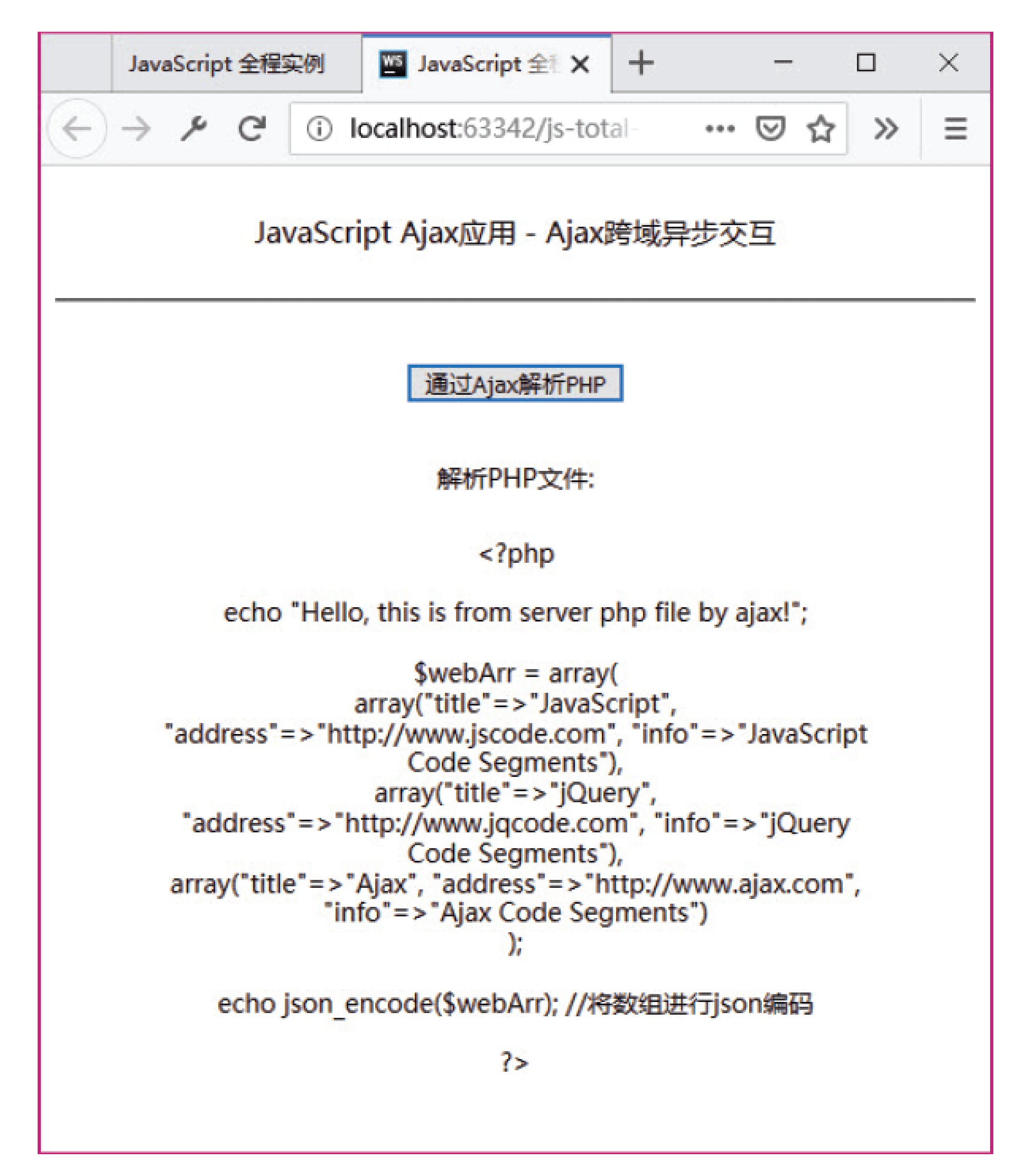


图 11.8　Ajax 跨域异步交互（无服务器环境）



图 11.9　Ajax 跨域异步交互（PHP 服务器环境）

第 12 章　React 开发

为了帮助读者开阔思路和增长见识，本章将详细介绍一下如今在 Web 前端开发领域非常火爆的 React 框架，具体内容涵盖了 React 框架的基础、使用入门以及 Web 前端应用开发，目的是让读者了解一下先进的 Web 前端开发技术的发展趋势。

12.1　React 概述

对于大多数读者而言，React 会感觉很陌生（可能不如 jQuery 那么耳熟能详）。其实，React 是大有来头的，最初是社交网站巨头 Facebook 公司的一个内部项目，是用来架构 Instagram 网站的。

现在感觉有点意思了吧！Instagram 网站就是在移动端中非常有名的、用于图片分享的社交应用，可以将用户随时随地抓拍的图片在移动终端设备（手机、平板电脑等）上彼此分享。Instagram 网站最初也是一家独立公司，于 2012 年被 Facebook 公司收购。

如同大多数的前端开发框架一样，React 最初的开发者也是不满意市面上已有的产品，所以决定自己搞一套出来打败前辈。React 的设计思想很独特、视角很新奇，应该说是有革命性的一类。总体而言，React 在性能上很出众，而代码逻辑却又非常简单，所以有越来越多的开发人员开始关注并使用该框架。

从本质上讲，React 框架主要用于构建前端 UI，其核心思想就是封装组件。各个组件维护自身的状态和 UI，每当状态变更时都会自动重新渲染整个组件，而不需要反复查找 DOM 元素后再重新渲染更改整个组件了。

下面简单概括一下 React 框架的主要特点（参考官方文档及网络资源）：

（1）声明式设计：React 采用声明范式，可以轻松描述应用。

（2）高效：React 通过对 DOM 的模拟，最大限度地减少与 DOM 的交互。

（3）灵活：React 可以与已知的库或框架很好地配合。

（4）使用 JSX 语法：JSX 是 JavaScript 语法的扩展，可以极大地提高 JS 运行效率。

（5）组件复用：通过 React 构建组件，使得代码易于复用，可在大型项目应用开发中发挥优势。

（6）单向响应的数据流：React 实现了单向响应的数据流，减少了重复代码，比传统数据绑定方式更简单。

通过 React 框架可以设计出功能强大的 Web 前端应用，后续将为读者介绍 React 开发的相关知识及案例。

备注：React 框架官方网址为 https://reactjs.org/。

12.2 第一个 React 应用

从本节开始，我们正式介绍如何使用 React 框架开发 Web 前端应用。React 应用开发涉及的知识面比较宽泛，如何切入着实费了一番脑筋。不过思前想后，还是觉得第一个 React 应用从基本的"Hello World"开始最合适。所谓"万变不离其宗"，指的也就是这个含义吧。

首先，要使用 React 框架就要先安装该框架。好在 React 框架可以直接通过引用的方式来使用，这里推荐两个比较好的 CDN 地址。

第一个是 Staticfile CDN 的 React CDN 库，具体如下：

```
<script src="https://cdn.staticfile.org/react/16.4.0/umd/react.development.js"></script>
<script src="https://cdn.staticfile.org/react-dom/16.4.0/umd/react-dom.development.js"></script>
<!-- 生产环境中不建议使用 -->
<script src="https://cdn.staticfile.org/babel-standalone/6.26.0/babel.min.js"></script>
```

第二个是官方提供的 CDN 库，具体如下：

```
<script src="https://unpkg.com/react@16/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
<!-- 生产环境中不建议使用 -->
<script src="https://unpkg.com/babel-standalone@6.26.0/babel.min.js"></script>
```

在如上的两个 CDN 库中，都引用了 react.min.js、react-dom.min.js 和 babel.min.js 这三个库文件，具体描述如下：

- react.min.js 是 React 框架的核心库。
- react-dom.min.js 提供与 DOM 相关的功能。
- babel.min.js 由 Babel 编译器提供，可以将 ES6 代码转为 ES5 代码，这样就能在不支持 ES6 的浏览器上执行 React 代码（注意，在生产环境中不建议使用）。

下面看一个使用 React 框架实现"Hello World"应用的代码实例。

【代码 12-1】（详见源代码目录 ch12-react-helloworld.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <script src="https://unpkg.com/react@16/umd/react.development.js"></script>
```

```
05      <script src="https://unpkg.com/react-dom@16/umd/
                                    react-dom.development.js"></script>
06      <!-- Don't use this in production: -->
07      <script src="https://unpkg.com/babel-standalone@6.26.0/babel.min.js">
                                                                    </script>
08      <title>JavaScript 全程实例</title>
09  </head>
10  <body>
11  <!-- 添加文档主体内容 -->
12  <header>
13      <nav>JavaScript React 开发 - 第一个 React 应用</nav>
14  </header>
15  <div id="root"></div>
16  <script type="text/babel">
17      ReactDOM.render(
18          <h1>Hello, world!</h1>,
19          document.getElementById('root')
20      );
21  </script>
22  </body>
23  </html>
```

关于【代码 12-1】的说明：

- 第 04 行、第 05 行和第 07 行代码分别引用了 React 框架所需的三个库文件（react.min.js、react-dom.min.js 和 babel.min.js）。
- 第 15 行代码通过<div id="root">标签元素定义了一个层，用于显示通过 React 框架渲染的文本内容。
- 第 17 ~ 20 行代码通过调用 React DOM 对象的 render()方法来渲染元素，具体内容如下：
  - 第 18 行代码定义了要引入的元素节点<h1>。
  - 第 19 行代码获取了页面中要渲染的元素节点<div id="root">。
  - 通过 ReactDOM.render()方法将<h1>元素节点渲染到页面的层<div id="root">元素节点中。

上面的第 17～20 行代码很直观，但是逻辑性不太强，尝试将其代码改写成如下形式就更容易理解了。

【代码 12-2】（详见源代码目录 ch12-react-dom-render.html 文件）

```
<script type="text/babel">
    const h1 = (<h1>Hello, world!</h1>);
    var root = document.getElementById('root');
    ReactDOM.render(h1, root);
</script>
```

关于【代码 12-2】的说明：

- 第 02 行代码通过 const 关键字定义了一个常量（h1），描述了要引入的元素节点<h1>。由此可见，在 React 框架中变量的使用非常灵活，可以将元素节点直接定义为变量形式来使用。
- 第 03 行代码获取了页面中要渲染的元素节点<div id="root">，保存在变量（root）中。
- 第 04 行代码调用 React DOM 对象的 render()方法，将 h1 元素节点渲染到 root 元素节点中。

下面可以分别使用 Firefox 浏览器运行测试【代码 12-1】和【代码 12-2】定义的 HTML 网页，具体效果（一致的）如图 12.1 所示。

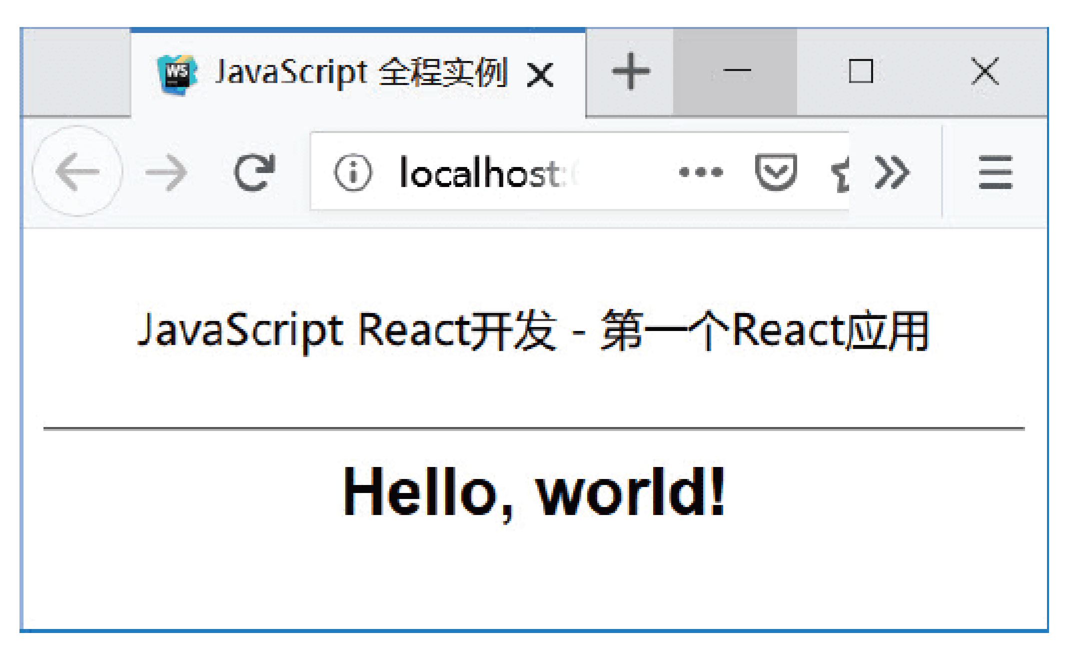


图 12.1　React 实现“Hello World”

如图 12.1 所示，页面中成功地显示了通过 React 框架渲染出的文本内容（Hello, world!）。

12.3　React 渲染更新元素

一般，React 元素都是不可变的，因此当元素被创建之后是无法改变其内容或属性的。如果打算更新界面元素，正确的办法就是创建一个新的元素对象，然后将其传入 ReactDOM.render()方法中来渲染更新元素。下面看一个具体的代码实例。

【代码 12-3】（详见源代码目录 ch12-react-dom-render-update.html 文件）

```
<div id="root"></div>
<script type="text/babel">
    /**
     * update time
     */
    function updateTime() {
        const helloworld = (<div>
            <h1>Hello, world!</h1>
            <h2>现在时间是 {new Date().toLocaleTimeString()}.</h2>
        </div>);
        var root = document.getElementById('root');
        ReactDOM.render(helloworld, root);
```

```
13      }
14      // TODO: define timer
15      setInterval(updateTime, 1000);
16  </script>
```

关于【代码 12-3】的说明：

- 第 01 行代码通过<div id="root">标签元素定义了一个层，用于显示通过 React 框架渲染的文本内容。
- 第 06 ~ 13 行代码定义了一个自定义方法 updateTime()，用于实现通过 React 渲染更新元素，具体内容如下：
  - 第 07 ~ 10 行代码通过 const 关键字定义了一个常量（helloworld），描述了要引入的容器节点<div>，包括一个<h1>标签元素（用于定义标题）和一个<h2>标签元素（定义标题内容）；同时，在<h2>标签元素内使用花括号定义了一个时间对象，用于获取当前时间。
  - 第 11 行代码获取了页面中要渲染的元素节点<div id="root">，保存在变量（root）中。
  - 第 12 行代码调用 React DOM 对象的 render()方法，将 h1 元素节点渲染到 root 元素节点中。
- 第 15 行代码使用 setInterval()方法设置了一个计时器，通过调用 updateTime()方法实现定时（1000ms）渲染更新元素。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 12.2 所示。

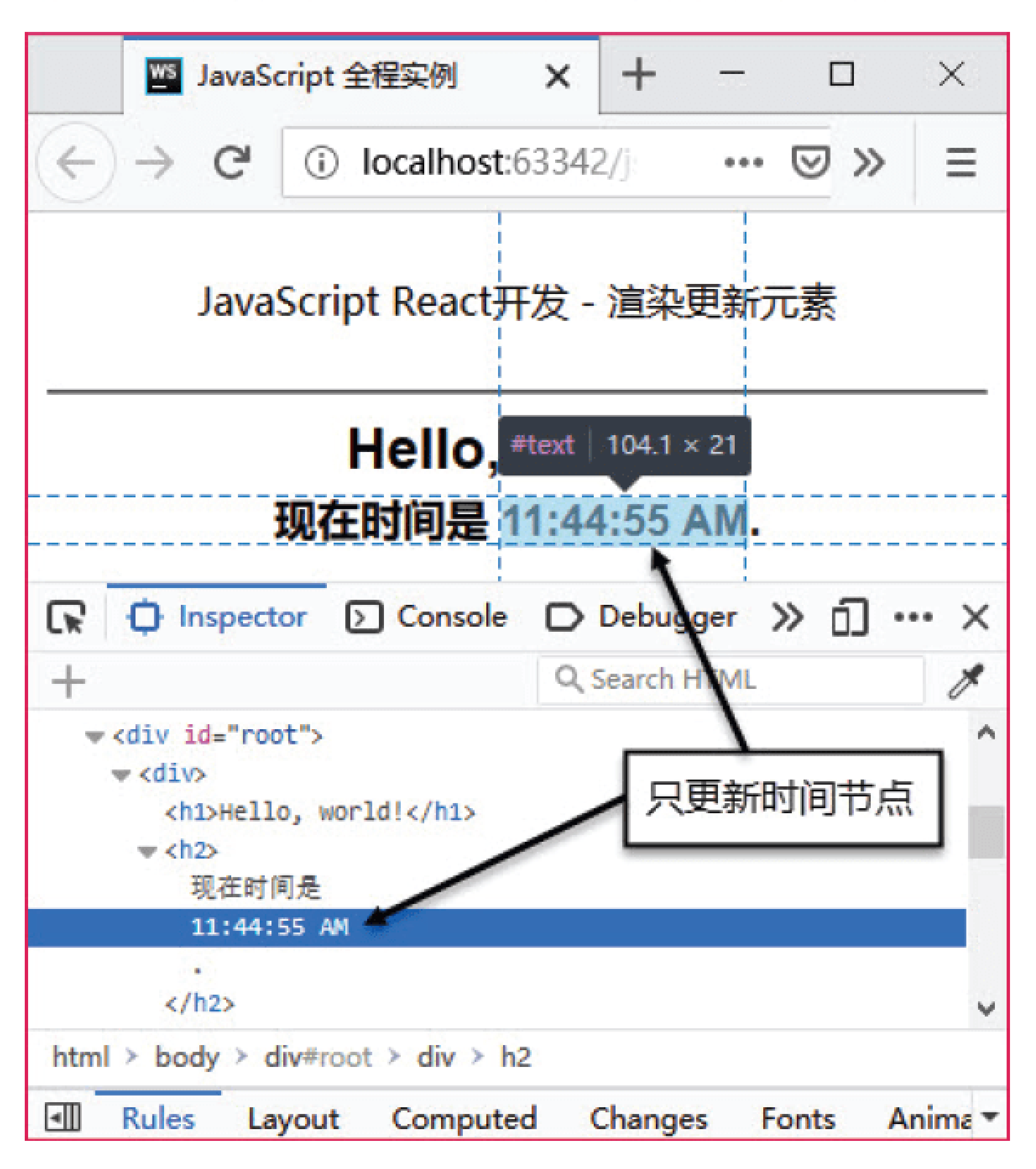


图 12.2　React 实现渲染更新元素

如图 12.2 中的箭头和标识所示，页面效果显示通过 React 渲染方式，仅仅只更新时间元素节点。

12.4 React 虚拟 DOM

React 的核心特性之一，就是可以通过创建虚拟 DOM 来避免重复过多的操作实际 DOM，从而实现对网页性能的提升。相信读者都知道，HTML 网页的骨架是由各类 DOM 元素构成的。但是，React 中定义的 DOM 并不是传统概念的 DOM，而是 React 用来在浏览器中创建元素组合的对象（理解为 DOM 组件似乎更直观），因此设计人员会将 React DOM 称为虚拟 DOM。

这里既然说到了虚拟 DOM，我们就先将其与实际 DOM 做一个比较，便于读者更好地理解 React DOM。说来也巧，在 HTML DOM 和 React DOM 中均定义了一个 createElement()方法，用来创建元素对象，只不过 HTML DOM 中的 createElement()方法创建的是一个实际 DOM 对象，而 React DOM 中的 createElement()方法创建的是一个虚拟 DOM。为了区别这个同名方法，先看一个通过 JavaScript 创建实际 DOM 的代码实例。

【代码 12-4】（详见源代码目录 ch12- js-createElement.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript React 开发 - 创建实际 DOM</nav>
</header>
<div id='id-div-react'></div>
<script type="text/babel">
    // TODO: get div
    var divReact = document.getElementById('id-div-react');
    // TODO: JavaScript DOM
    const jsSpan = document.createElement('span');
    const jsH3 = document.createElement('h3');
    jsH3.innerText = "JavaScript DOM";
    const jsP = document.createElement('p');
    jsP.innerText = "Create dom by JavaScript's createElement() func.";
    jsSpan.appendChild(jsH3);
    jsSpan.appendChild(jsP);
    divReact.appendChild(jsSpan);
</script>
</body>
</html>
```

关于【代码 12-4】的说明：

- 第 11 行代码通过<div id='id-div-react'>标签元素定义了一个层，是用于显示通过 JavaScript 创建实际 DOM 的容器。
- 第 16 ~ 23 行代码用于在层容器<div id='id-div-react'>中创建实际 DOM 节点，具体内容如下：
  - 第 16 行代码调用 Document 对象的 createElement()方法，创建了一个<span>元素节点。
  - 第 17 ~ 18 行代码调用 Document 对象的 createElement()方法，创建了一个<h3>元素节点，并定义了文本内容。
  - 第 19 ~ 20 行代码再次调用 Document 对象的 createElement()方法，创建了一个<p>元素节点，同样定义了文本内容。
  - 第 21 ~ 22 行代码分别调用 appendChild()方法将<h3>和<p>元素节点填充进<span>元素节点内。
  - 第 23 行代码再次调用 appendChild()方法，将<span>元素节点填充进层容器<div id='id-div-react'>内，从而实现创建实际 DOM 的操作。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 12.3 所示。

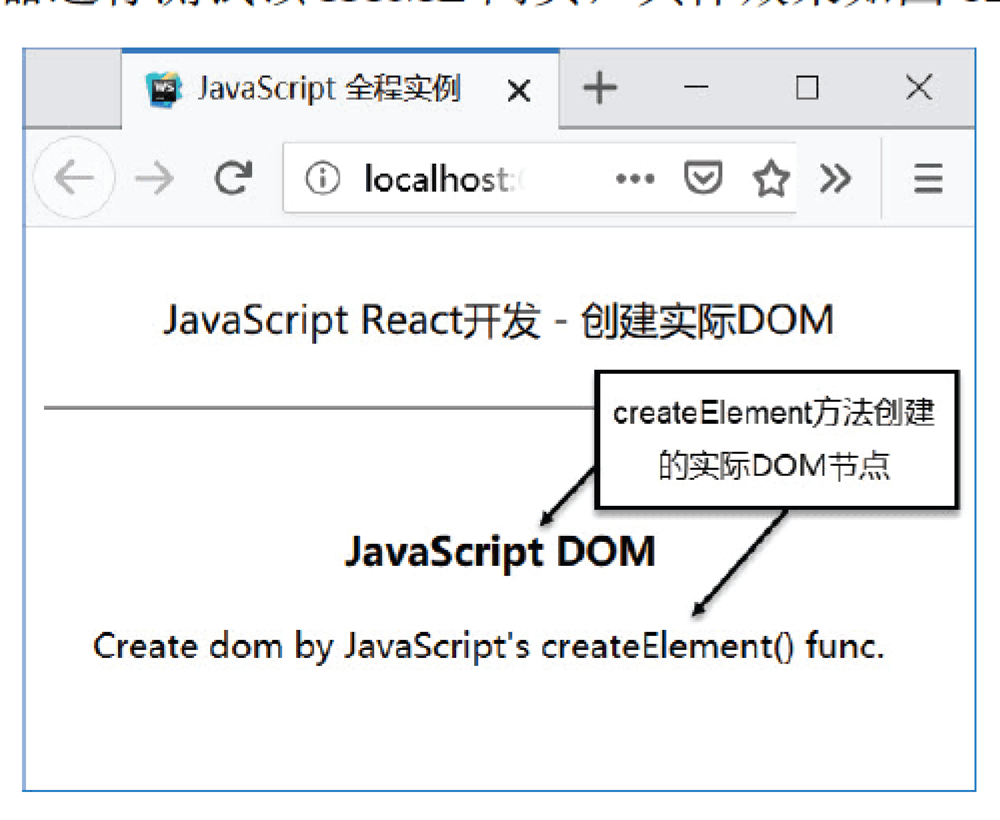


图 12.3　JavaScript 实现创建实际 DOM

下面看一个通过 React 创建虚拟 DOM 的代码实例。

【代码 12-5】（详见源代码目录 ch12-react-createElement.html 文件）

```
<div id='id-div-react'></div>
<script type="text/babel">
    // TODO: get div
    var divReact = document.getElementById('id-div-react');
    // TODO: React DOM
    const reactH3 = React.createElement("h3", {}, "React DOM");
    const reactP = React.createElement("p",{},"Create dom by React's
                                        createElement() func.");
    const reactSpan = React.createElement("span", {}, reactH3, reactP);
    ReactDOM.render(reactSpan, divReact);
</script>
```

关于【代码 12-5】的说明：

- 第 01 行代码通过<div id='id-div-react'>标签元素定义了一个层，是用于显示通过 React 创建虚拟 DOM 的容器。
- 第 06 ~ 09 行代码用于在层容器<div id='id-div-react'>中创建虚拟 DOM 节点，具体内容如下：
  - 第 06 行和第 07 行代码调用 React DOM 对象的 createElement()方法分别创建了一个<h3>元素节点（reactH3）和一个<p>元素节点（reactP），并相应定义了文本内容。
  - 第 08 行代码调用 React DOM 对象的 createElement()方法，创建了一个<span>元素节点（reactSpan），然后将刚刚创建的<h3>元素节点（reactH3）和<p>元素节点（reactP）填充进去。
  - 第 09 行代码调用 React DOM 对象的 render()方法，将<span>元素节点（reactSpan）渲染到层<div id='id-div-react'>容器中进行显示。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 12.4 所示。

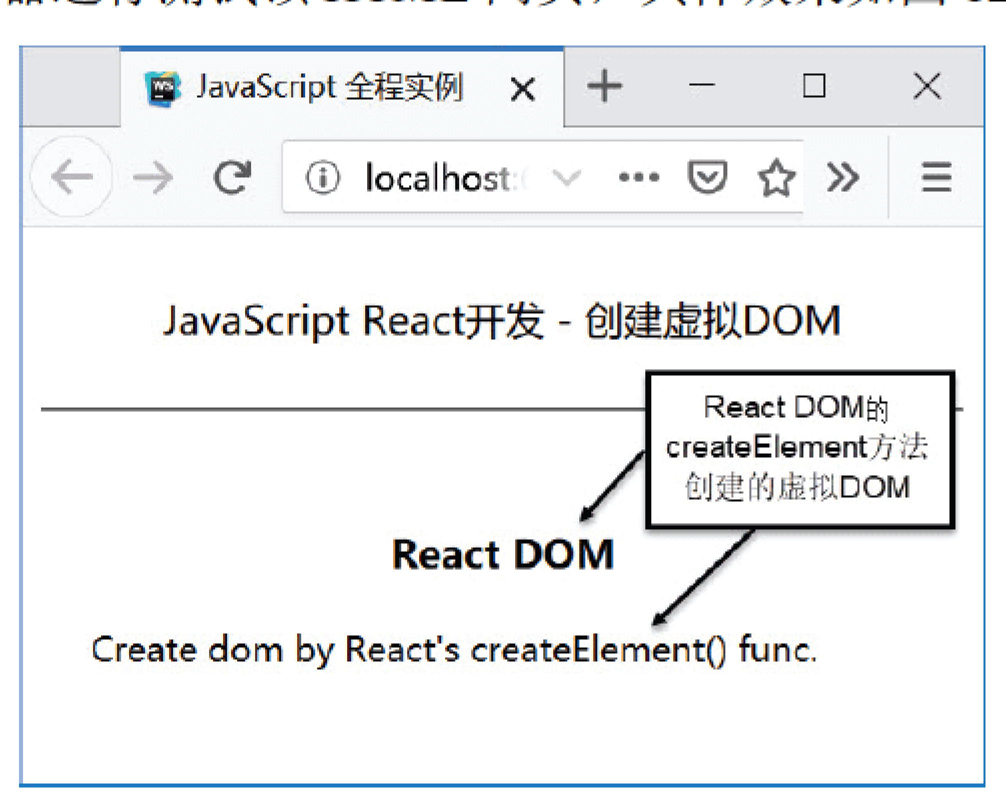


图 12.4　React 创建虚拟 DOM

如图 12.4 中的箭头和标识所示，页面中显示了通过 React DOM 的 createElement()方法渲染出来的虚拟 DOM。读者可以与图 12.3 中通过 JavaScript 创建的实际 DOM 进行对比，二者在显示效果上是等同的。

12.5　React JSX 初步

在 12.4 节中，我们介绍了如何通过 React DOM 的 createElement()方法创建虚拟 DOM，并将创建的虚拟 DOM 渲染到页面中的过程。但是，Facebook 的 React 研发团队还是觉得不够完美，于是就开发出来一种专属 React 的 JavaScript 语法——JSX。

所谓 JSX，其实就是 JavaScript XML 的缩写，直译过来就是基于 JavaScript 的 XML。JSX 看起来似乎是一种 XML 格式，其实本质上仍是 JavaScript 语言，只不过是将 js 脚本书写成 XML 格式。

JSX 作为一种标记语言，支持自定义属性，并具有可扩展性。同时，JSX 基本是专用于 React 开发的，如果打算在 HTML 页面输出 DOM 节点和内容，我们推荐使用 React 内置的 JSX 语法来实现。

另外，如果要在 React 中使用 JSX 语法，则必须引用“babel.js”文件来解析 JSX，同时<script>标签中必须改用“type="text/babel"”属性。原因在于使用“type="text/babel"”属性替换“type="text/javascript"”属性，浏览器内置的 JavaScript 解释器就不会解析里边的代码，转而由 babel 进行解析，从而避免 React 代码与原生 JavaScript 代码发生混淆。

介绍了这么多的概念和内容，下面让我们先看一下 JSX 的庐山真面目吧！其实，在前文【代码 12-1】中的第 18 行代码就已经使用了 JSX 语法，具体如下：

```
18        <h1>Hello, world!</h1>,
```

这里的<h1>标签元素就是通过 JSX 方式定义的，严格来说就是一个虚拟 DOM。

为了详细介绍 React JSX 的使用方法，下面先看一个将前文中【代码 12-5】改写成 JSX 方式的代码实例。

【代码 12-6】（详见源代码目录 ch12-react-jsx-intro.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <script src="https://unpkg.com/react@16/umd/react.development.js"></script>
    <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6.26.0/babel.min.js"></script>
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript React 开发 - JSX 初步</nav>
</header>
<div id='id-div-react'></div>
<script type="text/babel">
    // TODO: get div
    var divReact = document.getElementById('id-div-react');
    // TODO: React JSX
    const reactSpan = (
        <span>
            <h3>React </h3>
            <p>Create HTML DOM by React JSX.</p>
        </span>
    );
    ReactDOM.render(reactSpan, divReact);
</script>
```

```
27 </body>
28 </html>
```

关于【代码 12-6】的说明：

- 在第 15 行代码中，<script type="text/babel">标签内使用了 JSX 语法要求的 babel 属性，这一点要特别提醒读者注意一下。
- 第 19～24 行定义了一段完整的 JSX 代码，实现了一个虚拟 DOM 对象，具体内容如下：
  - ➢ 第 19～24 行代码通过 const 关键字定义了一个常量（reactSpan），该常量使用小括号包含了通过<span>、<h3>和<p>标签定义的元素组合。
  - ➢ 第 20～23 行代码定义的 HTML 标签符合 XML 格式，而常量（reactSpan）的定义完全符合 JavaScript 语法，因此该语法被称为 JSX。
- 第 25 行代码调用 React DOM 对象的 render()方法，将 JSX 代码渲染到页面中进行显示。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 12.5 所示。

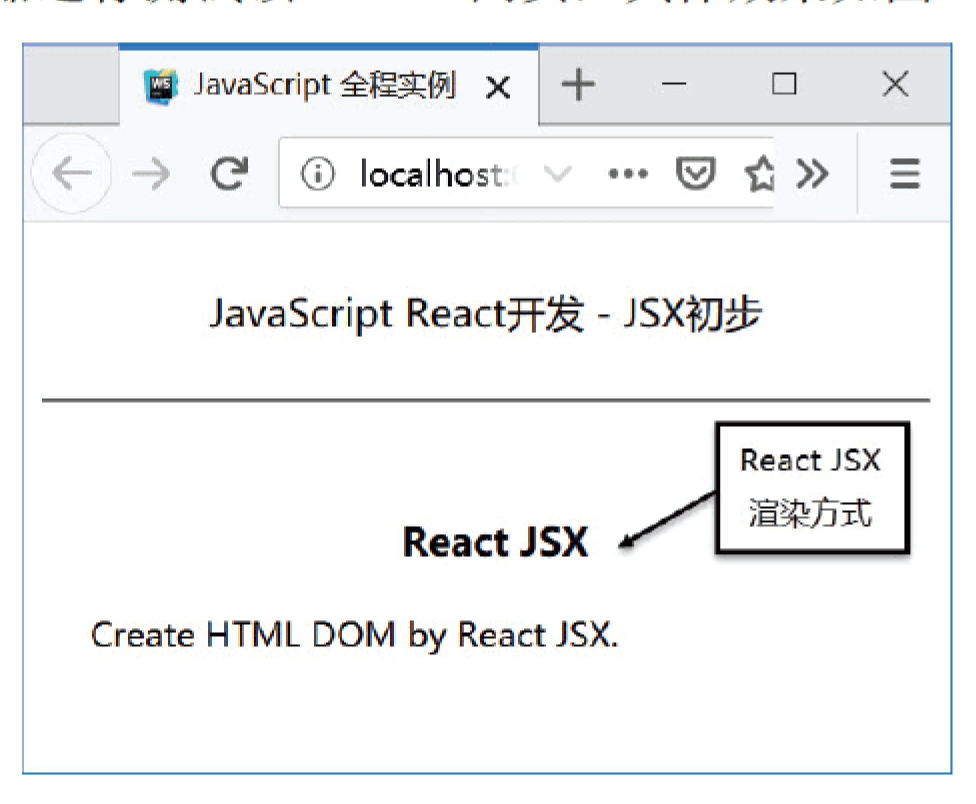


图 12.5 React JSX 初步

如图 12.5 中的箭头和标识所示，页面中显示了通过 React JSX 方式渲染的效果，与【代码 12-5】实现的功能是一致的。其实，通过 React JSX 方式定义的虚拟 DOM 最终也会转换为通过 createElement()方法实现的 DOM。

12.6 在 JSX 中使用 JavaScript 表达式

既然 React JSX 本质上使用的就是 JavaScript 语法，那么自然也可以使用 JavaScript 表达式。不过，在 React JSX 中使用 JavaScript 表达式要使用大括号“{}”括起来。React JSX 中的 JavaScript 表达式形式有很多种，我们先看一个最简单的 JavaScript 计算表达式的代码实例。

【代码 12-7】（详见源代码目录 ch12-react-jsx-cal-exp.html 文件）

```
<div id='id-div-react'></div>
<script type="text/babel">
```

```
03      // TODO: get div
04      var divReact = document.getElementById('id-div-react');
05      // TODO: React JSX
06      const reactSpan = (
07          <span>
08              <h3>JSX - JavaScript Calculate Expression</h3>
09              <p>Now calculate: 1 + 2 = {1+2}.</p>
10          </span>
11      );
12      ReactDOM.render(reactSpan, divReact);
13  </script>
```

关于【代码 12-7】的说明：

- 第 09 行代码中大括号“{1 + 2}”内定义的就是一个 JavaScript 计算表达式。React JSX 语法会将“1 + 2”的算术运算结果显示在页面中。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 12.6 所示。

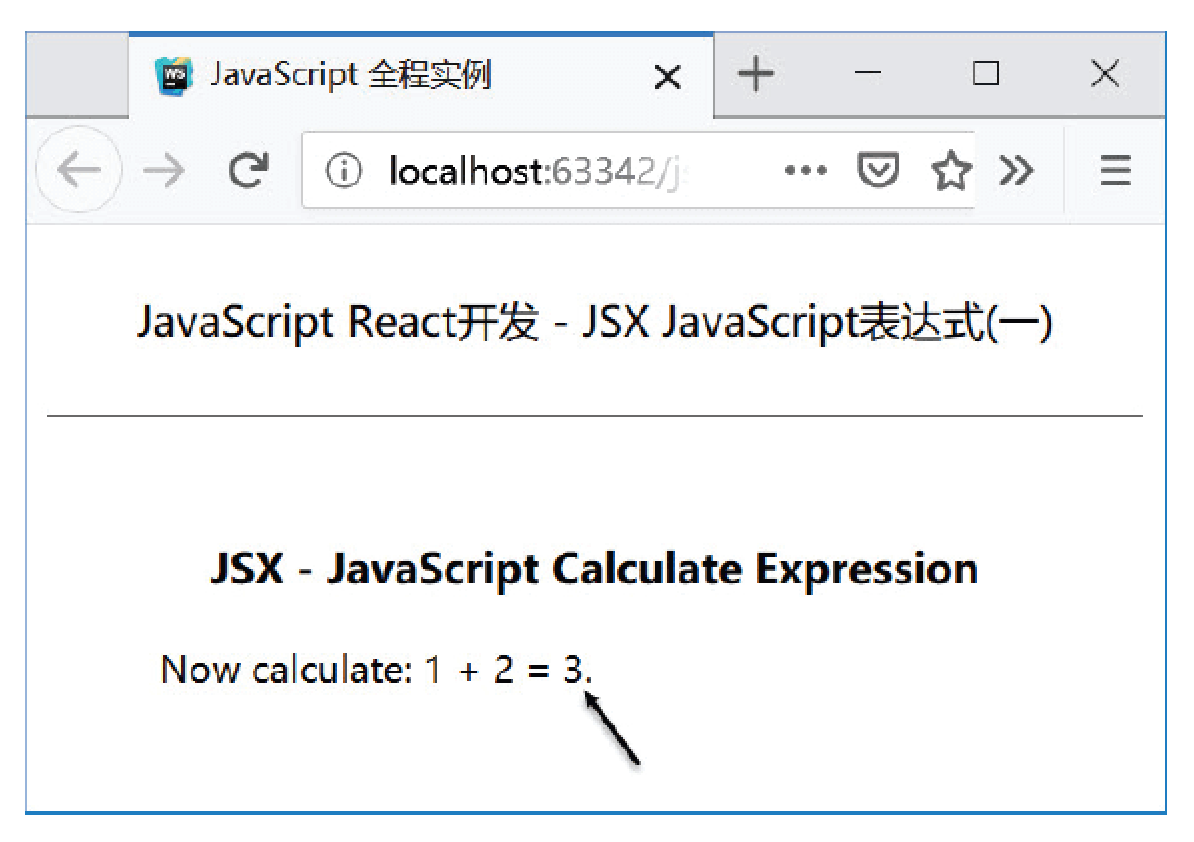


图 12.6　React JSX 中的 JavaScript 计算表达式

如图 12.6 中的箭头所示，页面中成功显示了表达式“1 + 2”的运算结果“3”。

React JSX 中的 JavaScript 表达式是无法使用条件语句（if）的，但是可以使用三元条件表达式。下面看一个具体的代码实例。

【代码 12-8】（详见源代码目录 ch12-react-jsx-tri-exp.html 文件）

```
<div id='id-div-react'></div>
<script type="text/babel">
    // TODO: get div
    var divReact = document.getElementById('id-div-react');
    // TODO: React JSX
    var i = 1;
    const reactSpan = (
```

```
08          <span>
09              <h3>JSX - JavaScript Triple Expression</h3>
10              <p>Return (i == 1 or i != 1) is : {i == 1 ? 'true' : 'false'}.</p>
11          </span>
12      );
13      ReactDOM.render(reactSpan, divReact);
14  </script>
```

关于【代码 12-8】的说明：

- 第 06 行代码定义了一个变量（i），初始化为数值 1 进行测试，然后变更为数值 2 进行测试。
- 第 10 行代码中大括号“{ i == 1 ? 'true' : 'false' }”内定义的就是一个 JavaScript 三元条件表达式。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 12.7 所示。

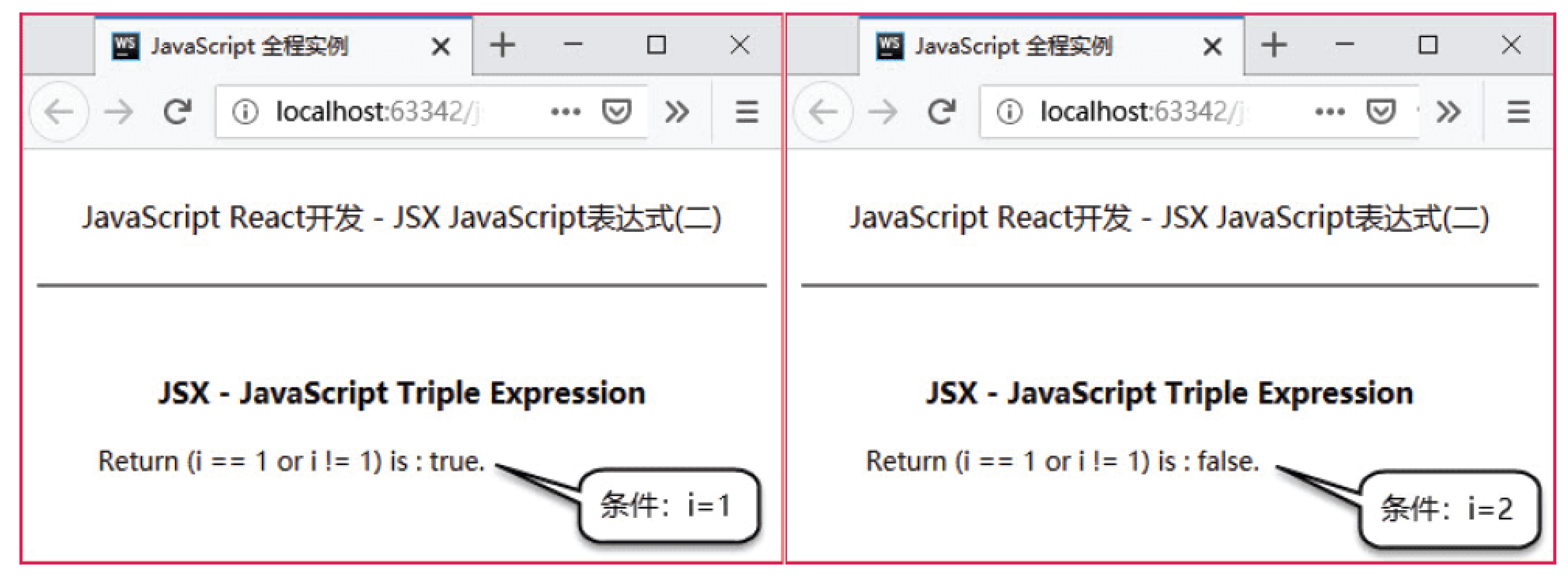


图 12.7　React JSX 中的 JavaScript 三元条件表达式

如图 12.7 中的箭头所示，左边页面中显示了初始条件“i=1”时的三元条件表达式的运算结果为“true”，右边页面中显示了初始条件“i=2”时的三元条件表达式的运算结果为“false”。

12.7　在 JSX 中使用 JavaScript 函数

本节在 JavaScript 表达式的基础上，进一步介绍在 JSX 中使用 JavaScript 函数。在 JSX 中调用 JavaScript 函数与表达式一样，同样是需要使用大括号“{ }”来引用的。下面我们看一个具体的代码实例。

【代码 12-9】（详见源代码目录 ch12-react-jsx-js-func.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <script src="https://unpkg.com/react@16/umd/react.development.js">
                                                                  </script>
```

```
05     <script src="https://unpkg.com/react-dom@16/umd/
                                     react-dom.development.js"></script>
06     <script src="https://unpkg.com/babel-standalone@6.26.0/babel.min.js">
                                     </script>
07     <title>JavaScript 全程实例</title>
08  </head>
09  <body>
10  <!-- 添加文档主体内容 -->
11  <header>
12     <nav>JavaScript React 开发 - 在 JSX 中使用 JavaScript 函数</nav>
13  </header>
14  <div id='id-div-react'></div>
15  <script type="text/babel">
16     // TODO: get div
17     var divReact = document.getElementById('id-div-react');
18     // TODO: define const obj
19     const userinfo = {
20        id: '007',
21        username: 'king',
22        gender: 'male'
23     };
24     function getUserinfo(ui) {
25        return ui.username + "[id:" + ui.id + "]'s gender is " +
                              ui.gender + ".";
26     }
27     // TODO: React JSX
28     const reactSpan = (
29        <span>
30           <h3>JavaScript Function in JSX</h3>
31           <p>{getUserinfo(userinfo)}</p>
32        </span>
33     );
34     ReactDOM.render(reactSpan, divReact);
35  </script>
36  </body>
37  </html>
```

关于【代码 12-9】的说明：

- 第 19 ~ 23 行代码定义了一个 JavaScript 常量对象（userinfo），并初始化了一组个人相关信息。
- 第 24 ~ 26 行代码定义了一个 JavaScript 函数（getUserinfo()），并通过第 25 行代码返回了一行文本信息（通过将个人相关信息所组合而成）。

- 第 28 ~ 33 行代码定义了一段完整的 JSX 代码，实现了一个虚拟 DOM 对象（常量 reactSpan）。其中，在第 31 行代码中调用了 JavaScript 函数（getUserinfo()）。
- 第 34 行代码调用 React DOM 对象的 render()方法，将虚拟 DOM 对象（常量 reactSpan）渲染到页面中进行显示。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 12.8 所示。



图 12.8　React JSX 中的 JavaScript 函数（一）

如图 12.8 中的箭头所示，页面中成功显示了通过 JavaScript 函数返回的个人信息。

React JSX 表达式还可以通过嵌入 JavaScript 函数中来运行，该方式的代码写法更加直接，更重要的是可以在 JSX 中使用条件语句（if）来进行逻辑判断。下面看一个具体的代码实例。

【代码 12-10】（详见源代码目录 ch12-react-jsx-exp-in-func.html 文件）

```
<div id='id-div-react'></div>
<script type="text/babel">
    // TODO: get div
    var divReact = document.getElementById('id-div-react');
    // TODO: define const obj
    const userinfo = {
        id: '007',
        username: 'king',
        gender: 'male'
    };
    // TODO: React JSX Exp in JavaScript Function
    function getUserinfo(ui) {
        if(ui) {
            return <span>
                    <h3>JSX Exp in JavaScript Function</h3>
                    <p>{ui.username + "[id:" + ui.id + "]'s gender is " +
                            ui.gender + "."}</p>
```

```
17                    </span>;
18          } else {
19              return <span>
20                    <h3>JSX Exp in JavaScript Function</h3>
21                    <p>empty user info.</p>
22                    </span>;
23          }
24      }
25      // TODO: React Render
26      ReactDOM.render(getUserinfo(userinfo), divReact);
27  </script>
```

关于【代码 12-10】的说明：

- 这段代码是在【代码 12-9】的基础上修改而成的，主要是将 JSX 代码（见第 14～17 行和第 19～22 行代码）以表达式的形式直接嵌入 getUserinfo()函数方法中来运行了。
- 另外，在第 13 行代码中加入了条件语句（if）来判断参数（ui）是否有效，然后根据判断结果选择相应的 JSX 代码。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 12.9 所示。

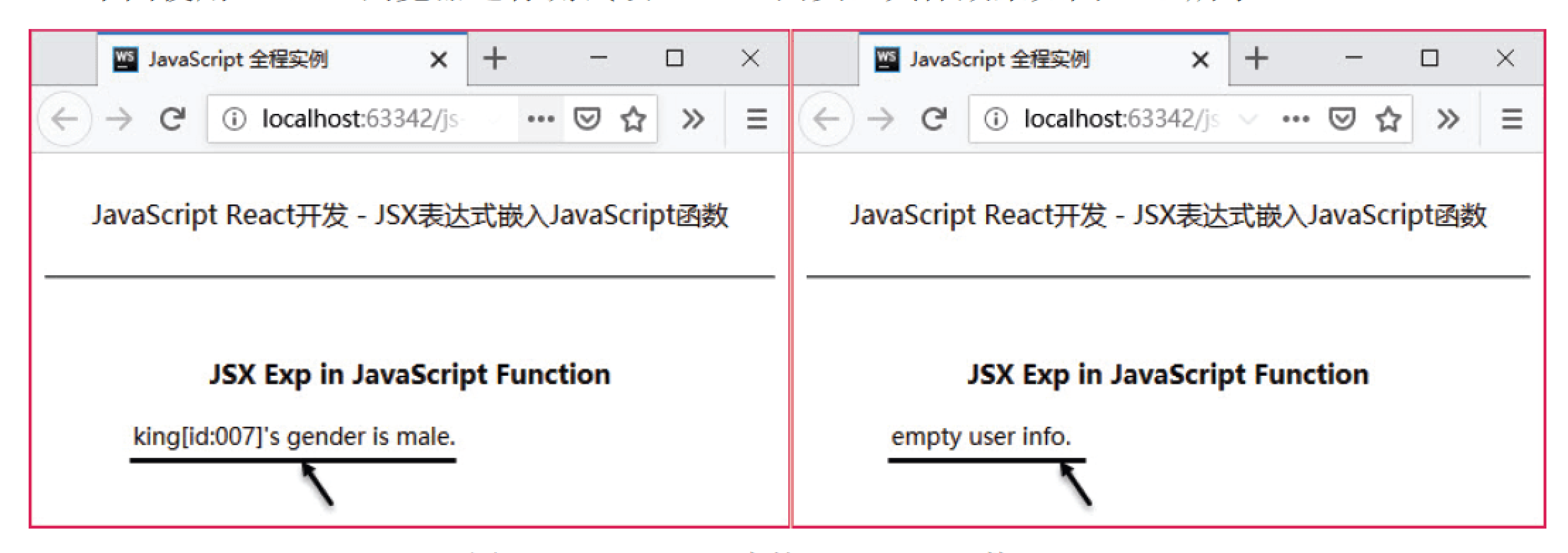


图 12.9　React JSX 中的 JavaScript 函数（二）

如图 12.9 中的箭头所示，左边页面中显示了参数（ui）有效时的页面效果，右边页面中显示了参数（ui）为空时的页面效果。

12.8　React Components 设计模式

React Components（组件）设计模式允许设计人员将用户界面的 UI 部分剥离出来，构成一个单独的部件。React Components（组件）在设计原理上保持了独立性和功能完整性，因此也就具有了可重复使用的特性，这也是 React 的性能优势之一。

在 React 中定义 Components（组件）一般有两种方式，分别是函数方式和 ES6 class 方式，这两种方式在功能效果上是等同的。下面我们分别介绍这两种方式的设计过程。

首先，看一个基于函数方式设计 React Components（组件）的代码实例。

【代码 12-11】（详见源代码目录 ch12-react-comp-func.html 文件）

```
01 <!DOCTYPE html>
02 <html>
03 <head>
04     <script src="https://unpkg.com/react@16/umd/react.development.js">
                                       </script>
05     <script src="https://unpkg.com/react-dom@16/umd/
                                       react-dom.development.js"></script>
06     <script src="https://unpkg.com/babel-standalone@6.26.0/babel.min.js">
                                       </script>
07     <title>JavaScript 全程实例</title>
08 </head>
09 <body>
10 <!-- 添加文档主体内容 -->
11 <header>
12     <nav>JavaScript React 开发 - React Components(函数方式)</nav>
13 </header>
14 <div id='id-div-react'></div>
15 <script type="text/babel">
16     // TODO: get div
17     var divReact = document.getElementById('id-div-react');
18     // TODO: function React Components
19     function ReactComp() {
20         return <span>
21             <h3>React Components(Func)</h3>
22             <p>This is a func React Components.</p>
23         </span>;
24     }
25     // TODO: define const
26     const elReactComp = <ReactComp/>;
27     ReactDOM.render(elReactComp, divReact);
28 </script>
29 </body>
30 </html>
```

关于【代码 12-11】的说明：

- 第 19～24 行通过一个自定义函数（ReactComp()）封装了一个虚拟 DOM 对象，具体说明如下:

 - 第 19 行代码声明的函数名（ReactComp）就是 React Components（组件）的名称（组件名）。
 - 第 20～23 行代码返回了一个通过<span>、<h3>和<p>标签元素组合而成的虚拟 DOM。
- 第 26 行代码通过 const 声明了一个常量（elReactComp），保存了以组件名（ReactComp）生成的自定义标签（<ReactComp/>）。特别要注意，原生 HTML 标签元素名可以小写字母开头，而 React 自定义组件名必须以大写字母开头；另外，自定义组件只能包含一个顶层标签，否则也会报错。
- 第 27 行代码调用 React DOM 对象的 render()方法，将自定义组件（<ReactComp/>）渲染到页面中进行显示。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 12.10 所示。

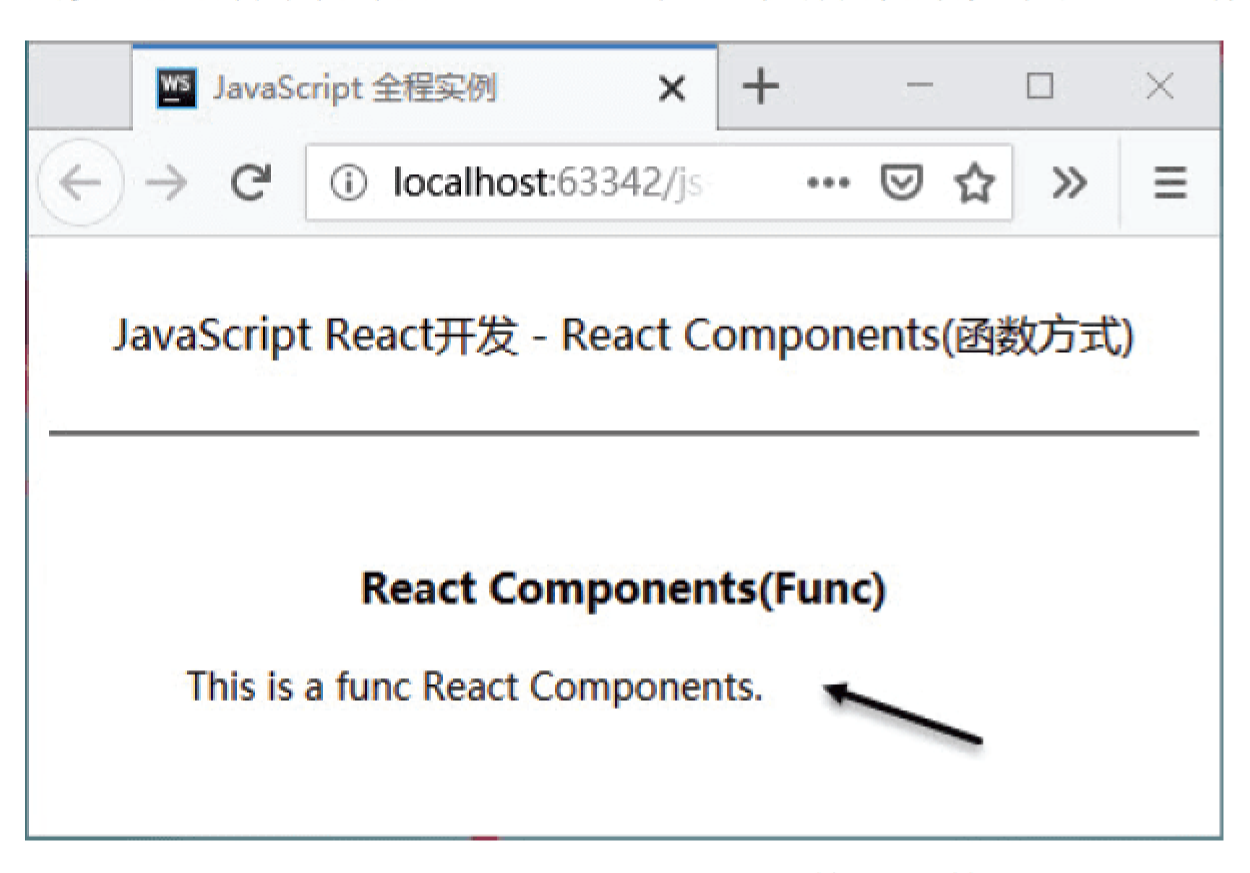


图 12.10　React Components（组件）函数方式

如图 12.10 中的箭头所示，页面中成功显示了通过 React Components（组件）函数方式渲染的效果。

除了函数方式外，React Components（组件）还可以使用 ES6 class 方式来生成。下面看一个基于 ES6 class 方式设计 React Components（组件）的代码实例。

【代码 12-12】（详见源代码目录 ch12-react-comp-class.html 文件）

```
<div id='id-div-react'></div>
<script type="text/babel">
    // TODO: get div
    var divReact = document.getElementById('id-div-react');
    // TODO: function React Components
    class ReactComp extends React.Component {
        render() {
            return <span>
                    <h3>React Components(ES6 class)</h3>
                    <p>This is a ES6 class React Components.</p>
                    </span>;
```

```
12          }
13      }
14      // TODO: define const
15      const elReactComp = <ReactComp/>;
16      ReactDOM.render(elReactComp, divReact);
17  </script>
```

关于【代码 12-12】的说明：

- 这段代码是在【代码 12-11】的基础上修改而成的，主要是将通过函数方式定义的组件（<ReactComp/>）改成通过 ES6 class 方式来定义。
- 第 06 ~ 13 行通过 class 关键字声明了一个 React Components（组件）类，具体说明如下：
  - 第 06 行代码定义类名为 ReactComp，继承自 React Component 对象。
  - 第 07 ~ 12 行代码通过 render 方法定义了一个通过<span>、<h3>和<p>标签元素组合而成的虚拟 DOM，然后通过 return 语句返回。
- 第 15 行代码通过 const 声明了一个常量（elReactComp），保存了以组件名（ReactComp）生成的自定义标签（<ReactComp/>）。
- 第 16 行代码调用 React DOM 对象的 render()方法，将自定义组件（<ReactComp/>）渲染到页面中进行显示。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 12.11 所示。

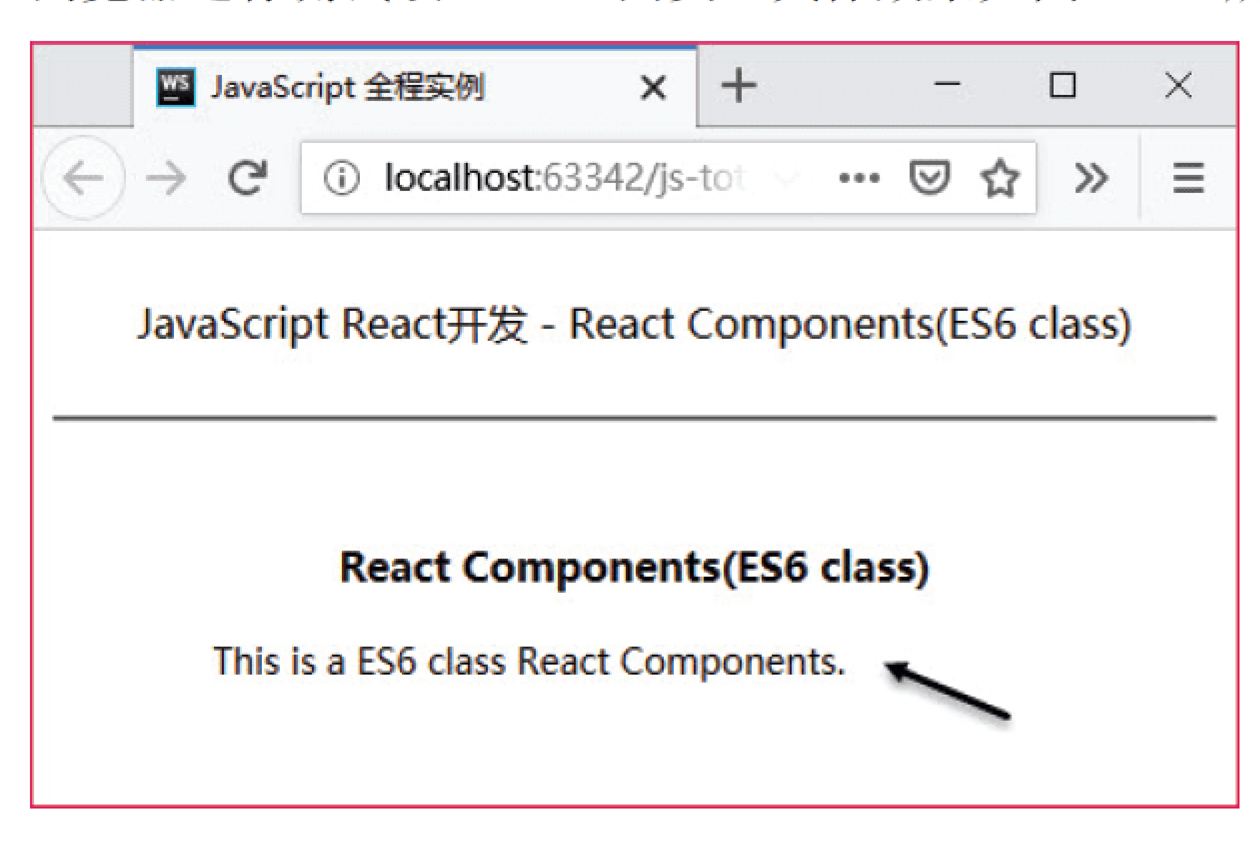


图 12.11　React Components（组件）ES6 class 方式

如图 12.11 中的箭头所示，页面中成功显示了通过 React Components（组件）ES6 class 方式渲染的效果。

12.9　React Components 参数

React Components（组件）还支持使用参数（properties），可以通过 props 对象来实现数据传递和返回。下面看一个具体的代码实例。

【代码 12-13】（详见源代码目录 ch12-react-comp-props.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <script src="https://unpkg.com/react@16/umd/react.development.js">
                                    </script>
    <script src="https://unpkg.com/react-dom@16/umd/
                                    react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6.26.0/babel.min.js">
                                    </script>
    <title>JavaScript 全程实例</title>
</head>
<style type="text/css">
    p.p-right {
        text-align: right;
    }
</style>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript React 开发 - React Components 参数</nav>
</header>
<div id='id-div-react'></div>
<script type="text/babel">
    // TODO: get div
    var divReact = document.getElementById('id-div-react');
    // TODO: function React Components with props
    function ReactComp(props) {
        return <span>
            <h3>React Components({props.name})</h3>
            <p>This is a webpage for {props.info}.</p>
            <p className={props.className}> --- by king.</p>
        </span>;
    }
    // TODO: define const
    const elReactComp = <ReactComp
                          name="props"
                          info="React Components Props"
                          className="p-right"/>;
    ReactDOM.render(elReactComp, divReact);
</script>
</body>
</html>
```

关于【代码 12-13】的说明：

- 这段代码是在【代码 12-11】的基础上修改而成的，主要是增加了参数 props 对象的使用。
- 第 24 ~ 30 行代码通过一个自定义函数（ReactComp()）封装了一个虚拟 DOM 对象，具体说明如下：
  - 第 24 行代码在声明的函数名（ReactComp）内定义了参数（props）对象。
  - 第 26 ~ 28 行代码分别通过参数（props）对象的“name”“info”和“className”属性将属性值传递到虚拟 DOM 中进行显示。
- 第 32 ~ 35 行代码通过 const 声明了一个常量（elReactComp），保存了以组件名（ReactComp）生成的自定义标签（<ReactComp/>）。在自定义标签中依次添加了“name”“info”和“className”属性，并定义了相应的属性值。这里要注意一点，原生 HTML 标签中的类属性名需要替换为 className，而不能使用原生的 class（因为 class 是 JavaScript 的保留字）。
- 第 36 行代码调用 React DOM 对象的 render()方法，将自定义组件（<ReactComp/>）渲染到页面中进行显示。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 12.12 所示。

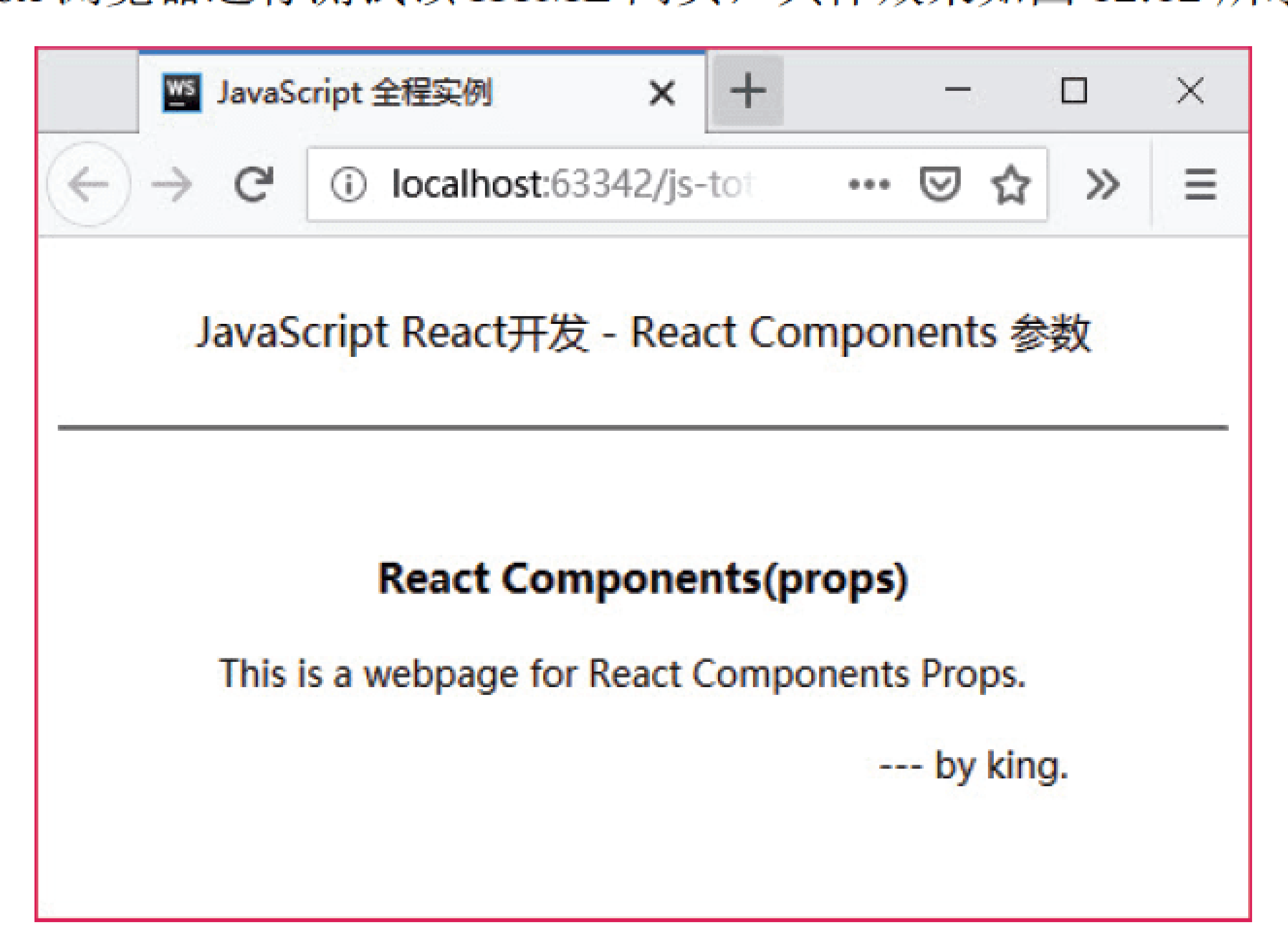


图 12.12　React Components（组件）参数（props）

如图 12.12 所示，页面中成功显示了通过 React Components（组件）参数（props）所渲染的效果。

12.10　React Components 复合

React Components（组件）的使用非常灵活，还可以通过创建多个组件来合成一个组件，一般称之为组件复合。React Components（组件）复合的思想是把组件按照不同的功能进行逻辑分割，使得业务逻辑更加清晰、管理更加高效。下面看一个关于 React Components（组件）复合的代码实例。

【代码 12-14】（详见源代码目录 ch12-react-comp-composing.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <script src="https://unpkg.com/react@16/umd/react.development.js">
                                             </script>
    <script src="https://unpkg.com/react-dom@16/umd/
                                    react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6.26.0/
                                    babel.min.js"></script>
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript React 开发 - React Components 复合</nav>
</header>
<div id='id-div-react'></div>
<script type="text/babel">
    // TODO: get div
    var divReact = document.getElementById('id-div-react');
    // TODO: function React Components - Title
    function Title(props) {
        return <h3>React Components {props.name}</h3>;
    }
    // TODO: function React Components - Content
    function Content(props) {
        return <p>This is a webpage for {props.info}.</p>;
    }
    // TODO: function React Components - Author
    function Author(props) {
        return <p className={props.className}>--- by King.</p>;
    }
    // TODO: function React Components by Composing
    function ReactComp() {
        return (
            <span>
                <Title name="Composing"/>
                <Content info="React Components by Composing"/>
                <Author className="p-right"/>
            </span>);
    }
```

```
39      // TODO: render
40      ReactDOM.render(<ReactComp/>, divReact);
41  </script>
42  </body>
43  </html>
```

关于【代码 12-14】的说明：

- 这段代码是在【代码 12-13】的基础上按照 React Components（组件）复合方式改写的。
- 第 19～21 行、第 23～25 行和第 27～29 行代码分别定义了三个组件(Title、Content 和 Author)。
- 第 31～38 行代码是 React Components(组件)复合的关键，通过一个自定义函数(ReactComp())封装了前面定义的三个组件（Title、Content 和 Author），具体说明如下：
  - ➢ 第 34 行代码通过以组件名（Title）生成的自定义标签（<Title/>）定义了一个虚拟 DOM，并添加了 “name” 属性；
  - ➢ 第 35 行代码通过以组件名（Content）生成的自定义标签（<Content/>）定义了一个虚拟 DOM，并添加了 “info” 属性；
  - ➢ 第 36 行代码通过以组件名（Author）生成的自定义标签（<Author/>）定义了一个虚拟 DOM，并添加了 “className” 属性。
- 第 40 行代码调用 React DOM 对象的 render()方法，将自定义组件（<ReactComp/>）渲染到页面中进行显示。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 12.13 所示。

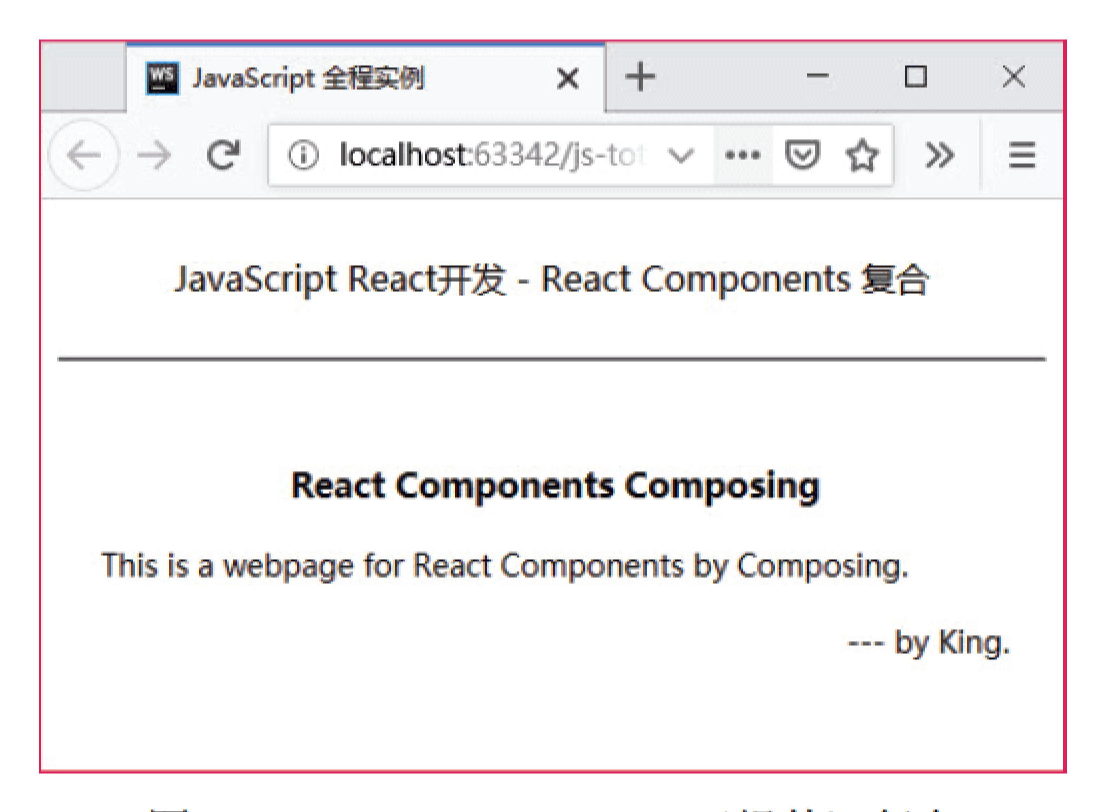


图 12.13　React Components（组件）复合

如图 12.13 所示，页面中成功显示了通过 React Components（组件）复合方式所渲染的效果。

12.11　React Components 状态

React 是把 Components（组件）视作一个状态机（State Machines）来设计的。为此，React 专门定义了一个 state（状态）属性用于更新 Components（组件）状态。在 React 应用中，用户交互

工作只需通过更新组件的 state（状态）就可以实现，根据更新后的 state（状态）来渲染 UI 并保持 UI 和数据的一致（不需要操作 DOM）。

下面先看一个关于 React Components（组件）state（状态）的代码实例。

【代码 12-15】（详见源代码目录 ch12-react-comp-state.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <script src="https://unpkg.com/react@16/umd/react.development.js">
                                                        </script>
    <script src="https://unpkg.com/react-dom@16/umd/
                                                        react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6.26.0/babel.min.js">
                                                        </script>
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript React 开发 - 组件状态(state)</nav>
</header>
<div id="id-div-react"></div>
<script type="text/babel">
    // TODO: get div
    var divReact = document.getElementById('id-div-react');
    // TODO: React Component (class)
    class ReactState extends React.Component {
        constructor(props) {
            super(props);
            console.log("state before init:");
            console.log(this.state);
            this.state = {name: "React Component State"};
            console.log("state after init:");
            console.log(this.state);
        }
        render() {
            return (
                <span>
                    <h3>React Component State</h3>
                </span>
            );
        }
```

```
35      }
36      // TODO: render
37      ReactDOM.render(<ReactState/>, divReact);
38  </script>
39  </body>
40  </html>
```

关于【代码 12-15】的说明：

- 第 19～35 行代码通过 ES6 class 方式创建一个 React Component 组件（ReactState），具体说明如下：
  - 第 20～27 行代码添加了一个类的构造函数（constructor）来初始化状态（state）。注意，状态（state）必须以 this 关键字的方式（this.state）调用，同时类组件应始终使用 super(props)来调用基础构造函数。
  - 第 24 行代码执行了初始化状态（state）属性的操作（{name: "React Component State"}），同时在初始化之前（第 22～23 行代码）和初始化之后（第 25～26 行代码）分别在浏览器控制台中输出状态（state），目的就是测试状态（state）的变化。
  - 第 28～34 行代码通过 render 方法定义了虚拟 DOM，用于显示 React Component 组件（ReactState）。
- 第 37 行代码调用 React DOM 对象的 render()方法，将自定义组件（<ReactState/>）渲染到页面中进行显示。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 12.14 所示。

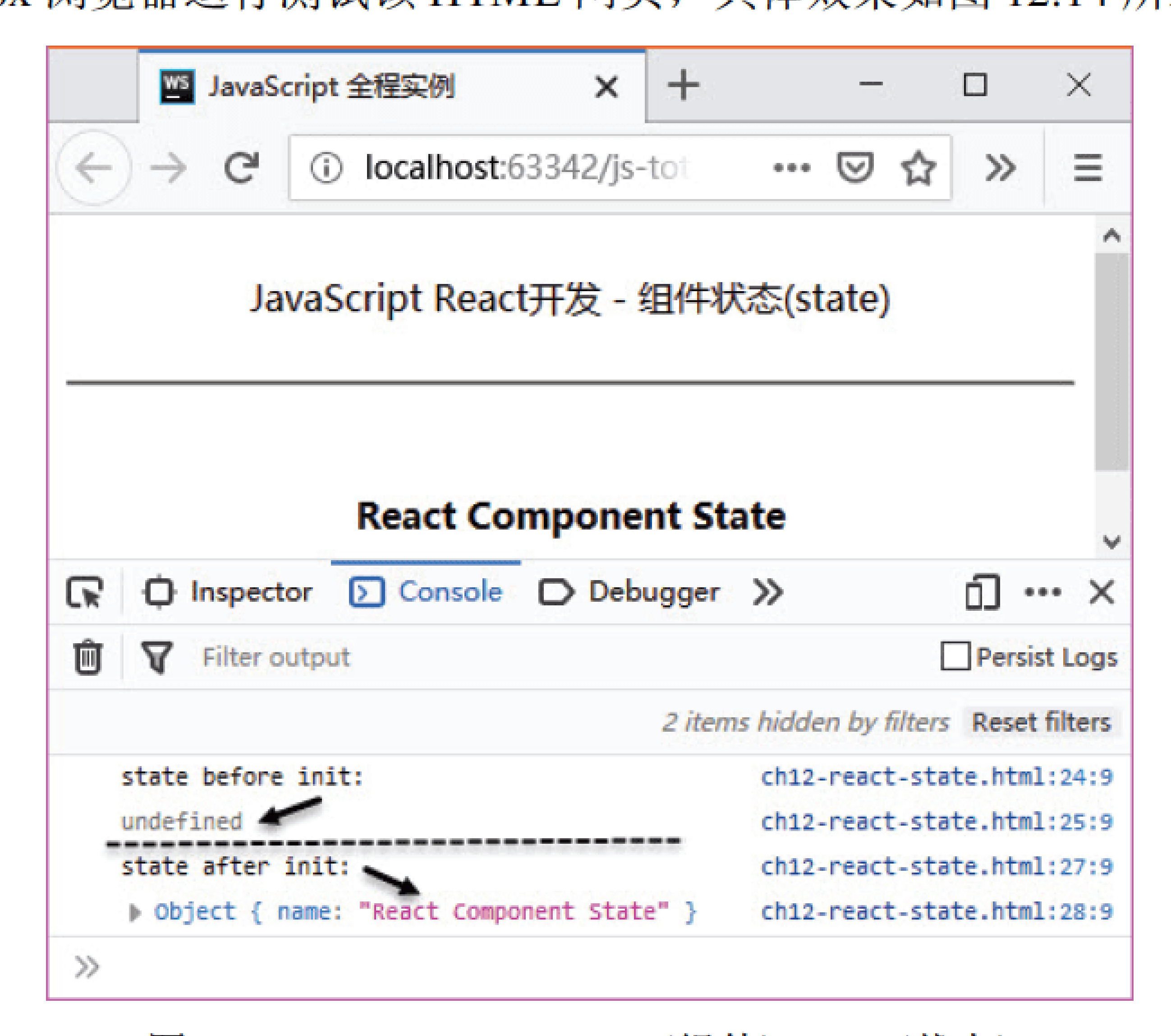


图 12.14　React Components（组件）state（状态）

如图 12.14 中的箭头所示，浏览器控制台中分别打印出了 state（状态）属性在初始化之前和初始化之后的值。

下面继续看一个如何使用 React Components（组件）state（状态）更新当前时间的代码实例。

【代码 12-16】（详见源代码目录 ch12-react-comp-state-date.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <script src="https://unpkg.com/react@16/umd/react.development.js">
                                    </script>
    <script src="https://unpkg.com/react-dom@16/umd/
                                    react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6.26.0/
                                    babel.min.js"></script>
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript React 开发 - 通过组件状态(state)更新时间</nav>
</header>
<div id="id-div-react"></div>
<script type="text/babel">
    // TODO: get div
    var divReact = document.getElementById('id-div-react');
    // TODO: React Component (class)
    class ReactState extends React.Component {
        constructor(props) {
            super(props);
            this.state = {date: new Date()};
        }
        render() {
            return (
                <span>
                    <h3>React Component State</h3>
                    <p>Now is {this.state.date.toLocaleTimeString()}.</p>
                </span>
            );
        }
    }
    // TODO: render
    ReactDOM.render(<ReactState/>, divReact);
</script>
</body>
</html>
```

关于【代码 12-16】的说明：

- 这段代码是在【代码 12-15】的基础上修改而成的，主要实现了更新当前时间的功能。
- 在组件类（ReactState）的构造方法中，第 22 行代码执行了初始化状态（state）属性的操作（{date: new Date()}），获取了时间 Date 对象（date）。
- 在组件类（ReactState）的 render 方法中，第 28 行代码通过状态（this.state）调用时间对象（date）更新了当前时间。
- 第 34 行代码调用 React DOM 对象的 render()方法，将自定义组件（<ReactState/>）渲染到页面中进行显示。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 12.15 所示。

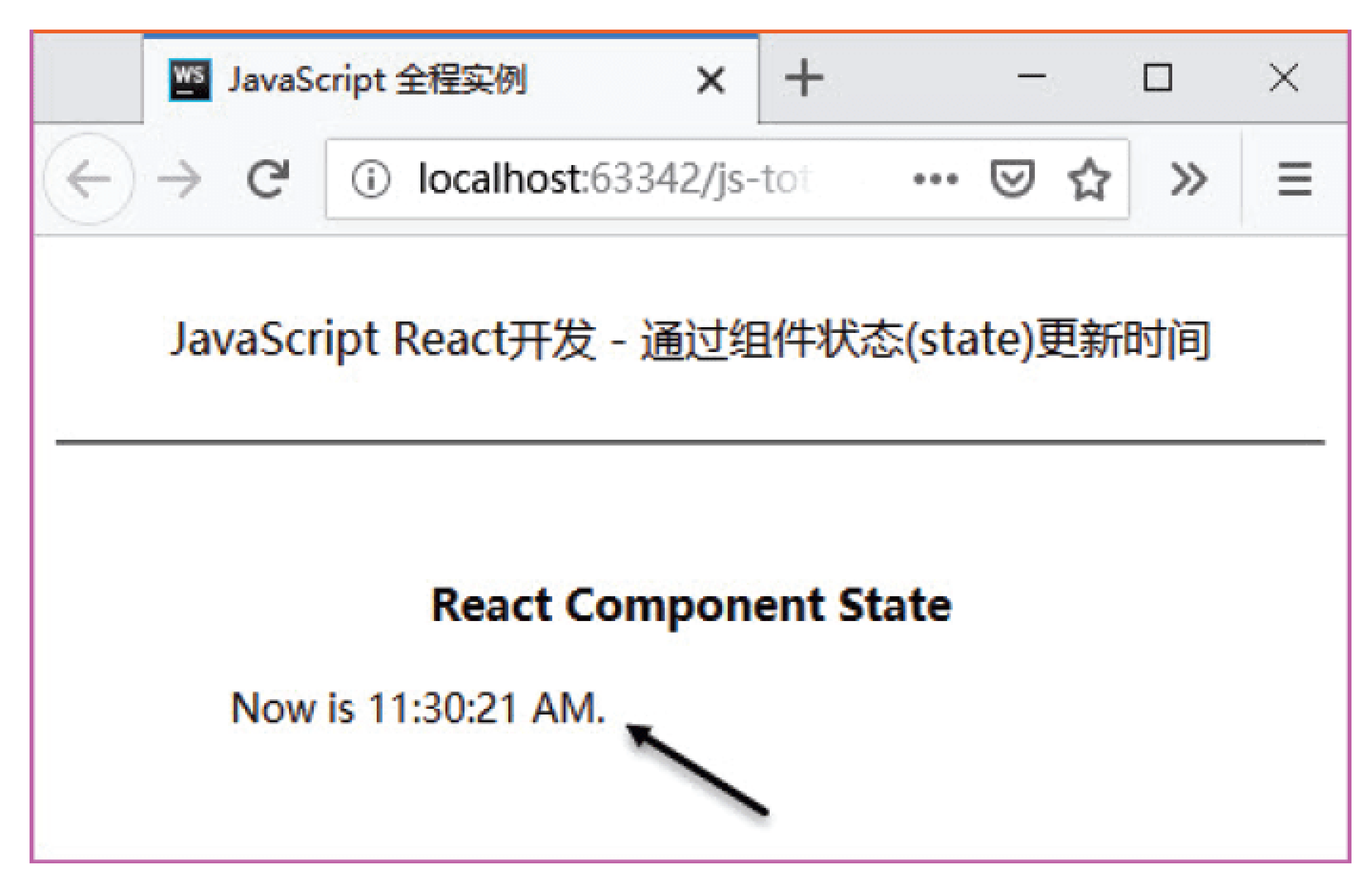


图 12.15　React Components（组件）通过 state（状态）更新时间

如图 12.15 中的箭头所示，浏览器中显示了通过 React Components（组件）state（状态）更新的当前时间。

12.12　React Components 生命周期

在前一节中，我们介绍了通过 React Components（组件）state（状态）更新当前时间的代码实例。读者应该会注意到，所更新的时间是静态的，而不是动态同步的。那么，如何实现时间每秒更新一次的效果呢？React Components（组件）设计了一个生命周期（LifeCycle）的方式，可以通过挂载和卸载计时器来实现该功能。

React 设计是推荐使用生命周期（LifeCycle）方式的，因为在实际项目开发中需要加载大量的组件，当这些组件不再使用后需要通过销毁来释放组件所占用的资源。生命周期（LifeCycle）方式提供了挂载（componentDidMount()）和卸载（componentWillUnmount()）这两个钩子方法来实现生命周期管理。

下面看一个通过 React Components（组件）生命周期方式来实现网页时钟功能的代码实例。

【代码 12-17】（详见源代码目录 ch12-react-comp-lifecycle.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <script src="https://unpkg.com/react@16/umd/react.development.js">
                                    </script>
    <script src="https://unpkg.com/react-dom@16/umd/
                                    react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6.26.0/
                                    babel.min.js"></script>
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript React 开发 - 生命周期(LifeCycle)</nav>
</header>
<div id="id-div-react"></div>
<script type="text/babel">
    // TODO: get div
    var divReact = document.getElementById('id-div-react');
    // TODO: React Component (class)
    class ReactLifeCycleClock extends React.Component {
        constructor(props) {
            super(props);
            this.state = {date: new Date()};
        }
        // TODO: Mount
        componentDidMount() {
            this.timerID = setInterval(
                () => this.timer(),
                1000
            );
        }
        // TODO: Unmount
        componentWillUnmount() {
            clearInterval(this.timerID);
        }
        // TODO: Timer
        timer() {
            this.setState({
```

```
38                  date: new Date()
39              });
40          }
41          // TODO: render
42          render() {
43              return (
44                  <span>
45                      <h3>React Component LifeCycle Clock</h3>
46                      <p>The current time is {this.state.date.
                                                toLocaleTimeString()}.</p>
47                  </span>
48              );
49          }
50      }
51      // TODO: render
52      ReactDOM.render(
53          <ReactLifeCycleClock/>,
54          divReact
55      );
56  </script>
57  </body>
58  </html>
```

关于【代码 12-17】的说明：

- 第 19 ~ 50 行代码通过 ES6 class 方式创建一个 React Component 组件（ReactLifeCycleClock），具体说明如下：
    - 第 20 ~ 23 行代码添加了一个类的构造函数（constructor）来初始化状态（state）。其中，第 22 行代码通过对状态（this.state）属性的操作（date: new Date()），定义了一个时间对象（date）。
    - 第 25 ~ 30 行代码定义了挂载钩子方法（componentDidMount()），并通过 setInterval()方法定义了一个计时器（timerID）。
    - 第 32 ~ 34 行代码定义了卸载钩子方法（componentWillUnmount()），并通过 clearInterval()方法清除了计时器（timerID）。
    - 第 36 ~ 40 行代码定义了一个用于更新时间的 timer()方法，该方法通过更新状态（state）属性实现同步时间的功能。
    - 第 42 ~ 49 行代码通过 render 方法定义了虚拟 DOM，用于显示 React Component 组件（ReactLifeCycleClock）。
- 第 52 ~ 55 行代码调用 React DOM 对象的 render()方法，将时钟自定义组件（<ReactLifeCycleClock/>）渲染到页面中进行显示。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 12.16 所示。

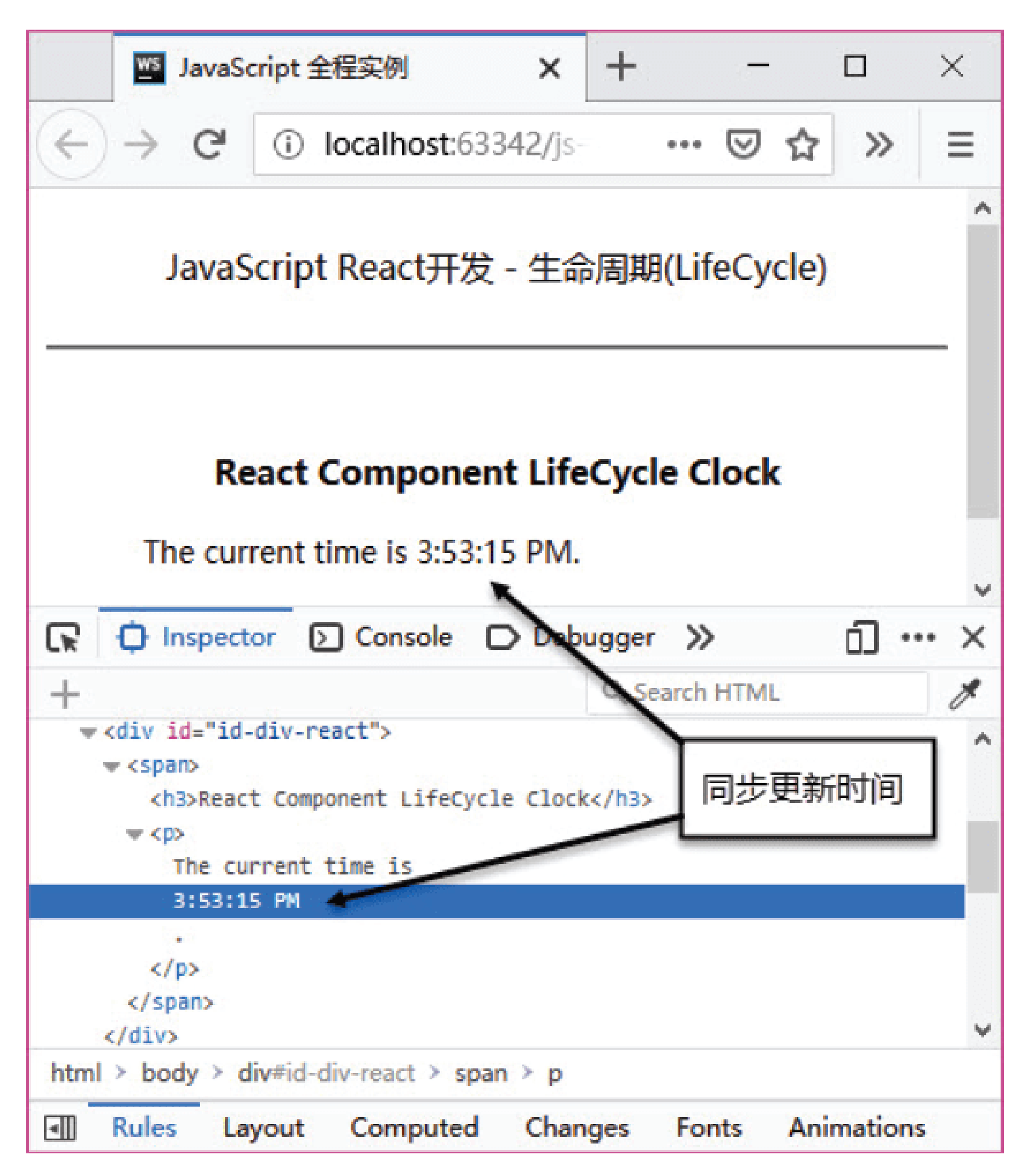


图 12.16　React Components（组件）生命周期（LifeCycle）实现时钟

如图 12.16 中的箭头和标识所示，页面中成功显示了通过 React Components（组件）生命周期（LifeCycle）方式实现的同步更新时钟的效果。

在 React Components（组件）中，无论是父组件还是子组件都不知道某个组件是有状态的还是无状态的，并且也不关心某组件是被定义为一个函数方式或是一个 ES6 class 方式的。这也就是为什么状态（state）通常会被称作是局部封装，除了拥有并设置它的组件外，其他组件都不可访问。

下面在【代码 12-17】的基础上添加一个 WaterfallDate 组件（函数方式），并在 props 属性中以多种方式接收 date 值。目的是让 props 属性不知道 date 值是来自 state 状态还是来自 date 属性输入，验证 React Components（组件）的独立性。在官方文档中，将下面的代码称为数据瀑布（Data Flow Down）方式。

【代码 12-18】（详见源代码目录 ch12-react-comp-lifecycle-waterfall-data.html 文件）

```
<script type="text/babel">
    // TODO: get div
    var divReact = document.getElementById('id-div-react');
    // TODO: React Component (func)
    function WaterfallDate(props) {
        return <p>Now is {props.date.toLocaleTimeString()}.</p>;
    }
    // TODO: React Component (class)
    class ReactLifeCycleClock extends React.Component {
        constructor(props) {
```

```
            super(props);
            this.state = {date: new Date()};
        }
        // TODO: Mount
        componentDidMount() {
            this.timerID = setInterval(
                () => this.timer(),
                1000
            );
        }
        // TODO: Unmount
        componentWillUnmount() {
            clearInterval(this.timerID);
        }
        // TODO: Timer
        timer() {
            this.setState({
                date: new Date()
            });
        }
        // TODO: render
        render() {
            return (
                <span>
                    <h3>React Component LifeCycle Clock</h3>
                    <p>The current time is
                              {this.state.date.toLocaleTimeString()}.</p>
                    <h3>LifeCycle Clock(WaterFall Data)</h3>
                    <WaterfallDate date={this.state.date}/>
                    <h3>LifeCycle Clock(Func)</h3>
                    <WaterfallDate date={new Date()}/>
                </span>
            );
        }
    }
    // TODO: render
    ReactDOM.render(
        <ReactLifeCycleClock/>,
        divReact
    );
</script>
```

关于【代码 12-18】的说明：

- 【代码 12-18】是在【代码 12-17】的基础上修改而成的，主要是在第 05 ~ 07 行代码中增加了一个 function 组件（WaterfallDate）。
- 同时，在第 32 ~ 43 行代码调用的 render()方法中增加了两种使用 WaterfallDate 组件的方式，具体如下：
  - 第 38 行代码是通过状态（state）属性获取的当前时间。
  - 第 40 行代码是直接通过人工方式（新建 Date 对象）获取的当前时间。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 12.17 所示。

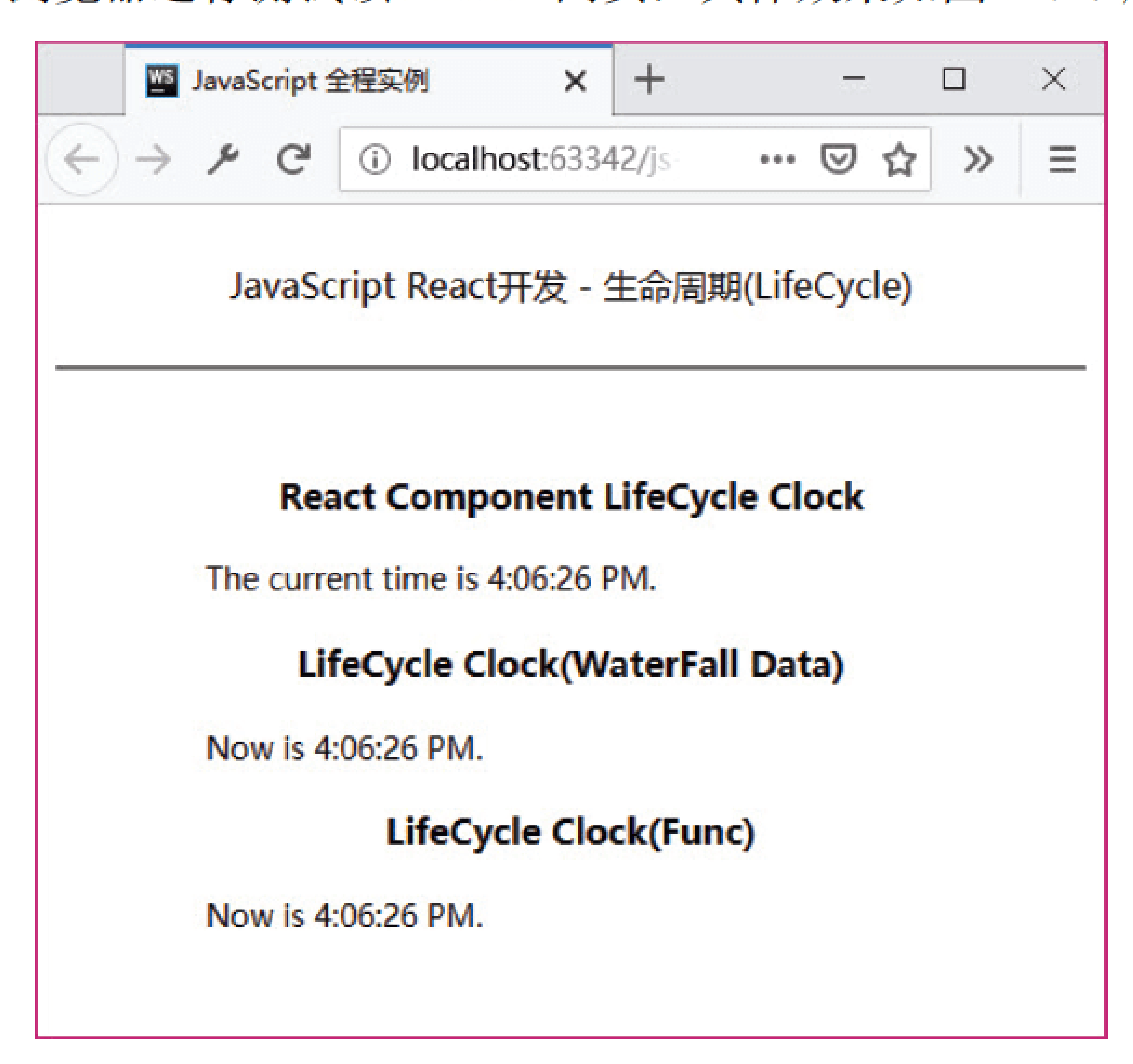


图 12.17　React Components（组件）生命周期（LifeCycle）实现时钟（数据瀑布方式）

如图 12.17 所示，数据瀑布方式类似于自顶向下的单向数据流，不论任何状态都始终由某些特定组件所有，并且从该状态导出的任何数据只能影响到下方的组件。

第 13 章　Vue.js 开发

本章介绍在 Web 前端开发领域非常火爆的 Vue.js 框架（一款用来构建用户界面的渐进式 Web 框架）。本章的内容主要包括 Vue.js 框架概述、渲染原理、安装使用、模板语法、条件循环和事件属性等基础知识。

13.1　Vue.js 概述

Vue.js（/vju: /，发音类似于英文单词 view）是一套构建用户界面的渐进式 Web 框架，其设计思想就是通过以数据驱动和组件化视图来构建用户界面。

Vue.js 框架采用自底向上增量开发的设计模式，框架的核心库专注于视图层设计，这样就易于与其他框架或已有项目进行整合。同时，Vue.js 与单文件组件或 Vue.js 生态系统扩展库相结合使用时，也完全能够为复杂的单页应用提供驱动。

对于 Vue.js 框架，业内有过这样的评价：“Vue.js 兼具 Angular.js 和 React 的优点，并最大程度摒弃了它们的缺点”。同时，Vue.js 框架之所以深受广大国内程序员的推崇，因其作者是一位地道的中国人（尤雨溪，Evan You），这也是国人在开源世界中的骄傲。

本质上，Vue.js 框架主要是基于数据驱动来专注于视图组件设计的，其提供了更加简洁、更易于理解的 API，使得设计人员能够快速地上手并使用。在使用 Vue.js 框架之前，建议先掌握以下各方面的编程知识：

（1）HTML 网页设计。
（2）CSS3 层叠样式代码。
（3）JavaScript 脚本语言开发。

由此可见，学习 Vue.js 框架要求门槛相对容易，相信大多数的读者都具有以上编程语言的开发经验。

通过 Vue.js 框架可以设计出功能强大的 Web 视图应用，本章将为读者介绍 Vue.js 开发的相关知识及案例。

备注：Vue.js 框架官方网址为 https://cn.vuejs.org。

13.2　第一个 Vue.js 应用

从本节开始，我们正式介绍如何使用 Vue.js 框架开发 Web 前端应用。正所谓“万变不离其宗”，第一个 Vue.js 应用还是从基本的 “Hello World” 开始最合适。

首先，使用 Vue.js 框架要先引用 Vue.js 库文件（区分开发版本和生产版本），这里推荐使用如下 CDN 地址。

开发版本的 React CDN 库如下：

```
<!-- 开发环境版本，包含了有帮助的命令行警告 -->
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
```

生产版本的 CDN 库如下：

```
<!-- 生产环境版本，优化了尺寸和速度 -->
<script src="https://cdn.jsdelivr.net/npm/vue"></script>
```

当然，读者也可以将 vue. js 或 vue.min.js 库文件下载到本地来引用。另外，如果想使用 Vue Devtools 进行调试，就必须使用开发版本的库文件。

下面看一下使用 Vue.js 框架实现 “Hello World” 应用的代码实例。

【代码 13-1】（详见源代码目录 ch13-vue-js-hello.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <script src="js/vue.js"></script>
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript Vue.js 开发 - Hello</nav>
</header>
<div id='id-div-vue'>
    <h3 id="app">{{ message }}</h3>
</div>
<script>
    var app = new Vue({
        el: '#app',
        data: {
            message : "Hello Vue.js!"
```

```
20            }
21        });
22    </script>
23    </body>
24    </html>
```

关于【代码 13-1】的说明：

- 第 04 行代码引用了 Vue.js 框架所需的库文件（vue.js）。这里引用的是开发版的库文件，当然也可以引用压缩版的库文件（vue.min.js，不过无法使用 Vue Devtools 调试功能）。
- 第 12～14 行代码通过<div>标签元素定义了一个层，其中第 13 行代码定义的<h3 id="app">标签元素用于显示通过 Vue.js 框架渲染的文本内容。
- 第 16～21 行代码就是 Vue.js 的核心，采用模板语法定义的声明式数据渲染 DOM 系统，具体内容如下：
  - ➢ 第 16 行代码通过 Vue 构造器（注意使用 new 关键字）创建了一个 Vue 对象（app）。
  - ➢ 第 17 行代码通过“el”参数指定将要渲染的 DOM（通过 id 属性）。
  - ➢ 第 18～20 行代码通过“data”参数定义属性，就是要渲染的数据内容。其中，在第 19 行代码中定义了一个“message”属性，其属性值将会渲染到第 13 行代码中对应的“{{message}}”位置，特别注意这里要将“message”放进双大括号中来使用。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 13.1 所示。

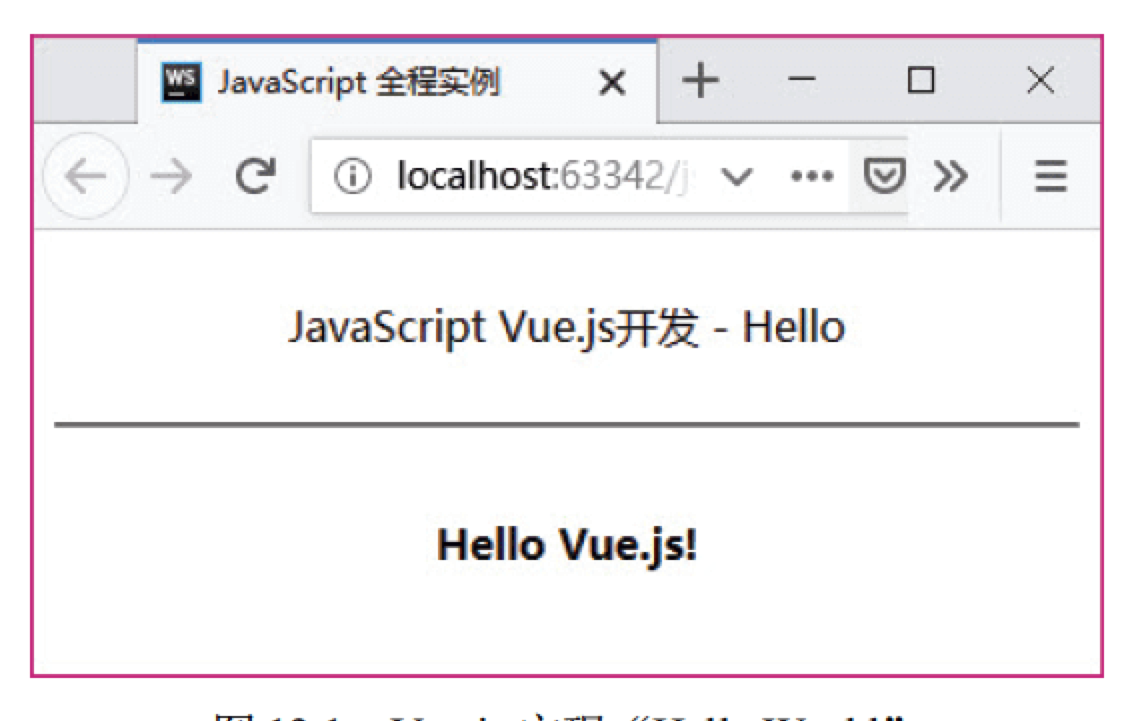


图 13.1　Vue.js 实现“Hello World”

如图 13.1 所示，页面中成功显示了通过 Vue.js 框架渲染出的文本内容（Hello, Vue.js!）。用很少的代码就实现了渲染功能，是不是很激动呢？

Vue.js 应用看起来似乎异常简单，其实 Vue.js 框架在底层做了大量工作。最主要的一点就是数据和 DOM 已经在内部进行了绑定，所有数据更新都是响应式的。下面我们通过浏览器的 JavaScript 控制台来验证一下 Vue.js 框架的这个特性，具体操作步骤如图 13.2 所示。

如图 13.2 中的箭头和标识所示，我们首先在左边页面的浏览器 JavaScript 控制台中输入属性名称“app.message”（对应【代码 13-1】的第 19 行代码），然后输入新修改的属性值（“Hello, King!”）后按回车键确认。接着在右边页面中成功显示了同步更新的文本内容（“Hello, King!”），证明了 Vue.js 框架的数据响应式更新功能。感兴趣的读者可以按照上面的步骤操作体验一下。

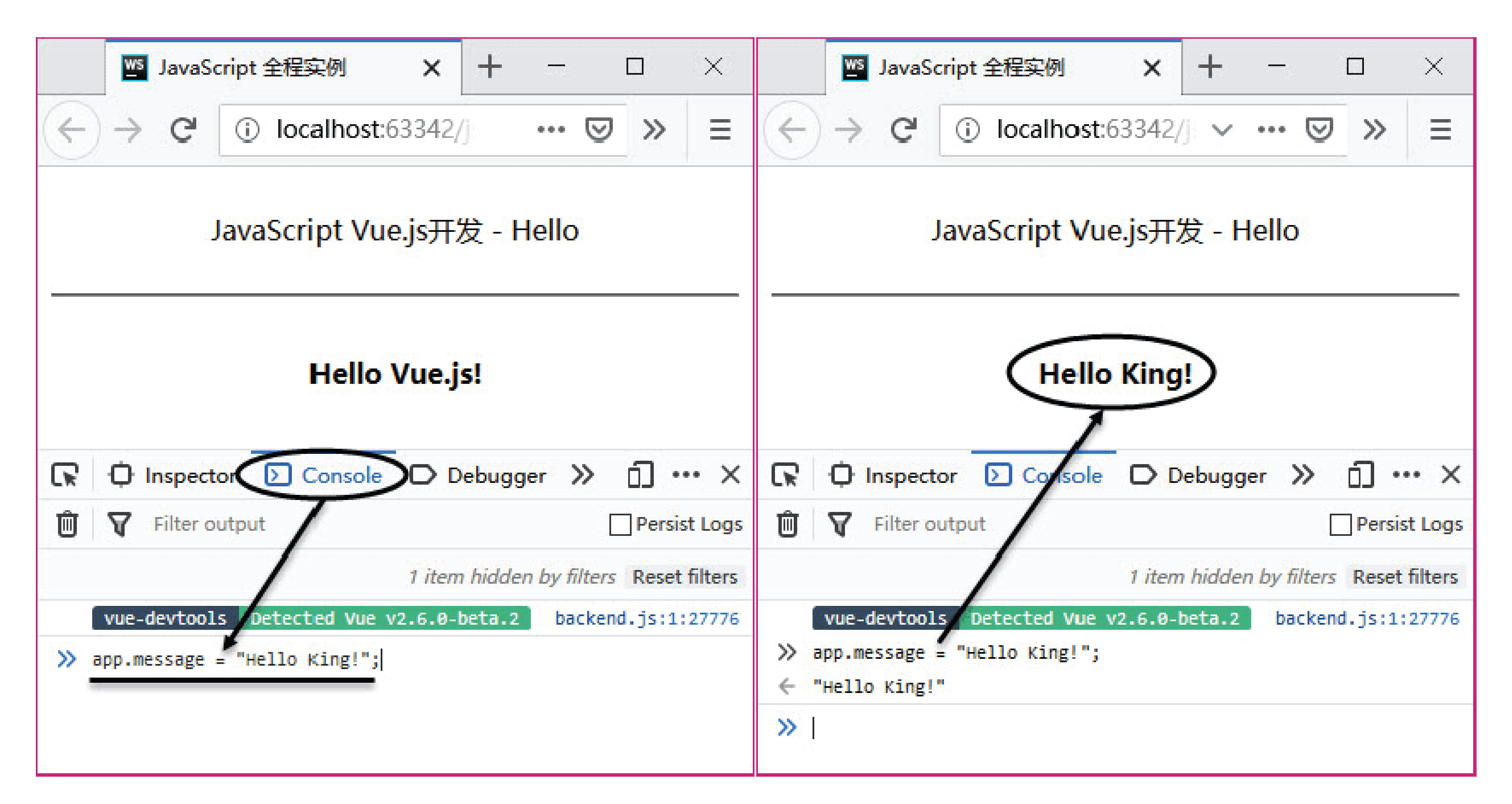


图 13.2　Vue.js 实现数据响应式功能

13.3　Vue.js 构造器

这一节我们介绍关于 Vue.js 框架构造器的内容。所谓构造器，就是创建 Vue 渲染的容器，具体代码形式如下：

```
<!-- 创建 Vue 构造器 -->
var app = new Vue {
  el:
  data: {   }
  method: {   }
}
```

其中，“el”参数指定要渲染的目标 DOM，“data”参数用于定义属性，“method”参数用于定义函数方法。

下面看一个关于使用 Vue 构造器的基本代码实例。

【代码 13-2】（详见源代码目录 ch13-vue-js-vue.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <script src="js/vue.js"></script>
    <title>JavaScript 全程实例</title>
</head>
</style>
<body>
<!-- 添加文档主体内容 -->
<header>
```

```
11      <nav>JavaScript Vue.js 开发 - Vue 构造器</nav>
12  </header>
13  <div id='id-div-vue'>
14      <h1>{{ title }}</h1>
15      <h3>{{ content }}</h3>
16      <p>{{ all() }}</p>
17  </div>
18  <script type="text/javascript">
19      var app = new Vue({
20          el: '#id-div-vue',
21          data: {
22              title : "Vue 构造器",
23              content : "通过 Vue 构造器创建应用, 实现页面渲染功能."
24          },
25          methods: {
26              all : function () {
27                  return this.title + " : " + this.content;
28              }
29          }
30      });
31  </script>
32  </body>
33  </html>
```

关于【代码 13-2】的说明：

- 第 20 行代码通过“el”参数指定将要渲染的 DOM（id = 'id-div-vue'）。
- 第 21 ~ 24 行代码通过“data”参数定义了一组属性（title 和 content），描述了要渲染的文本内容。
- 第 25 ~ 29 行代码通过“methods”参数定义了一个函数方法（all），通过 return 语句返回了两个属性（title 和 content）内容的组合。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 13.3 所示。

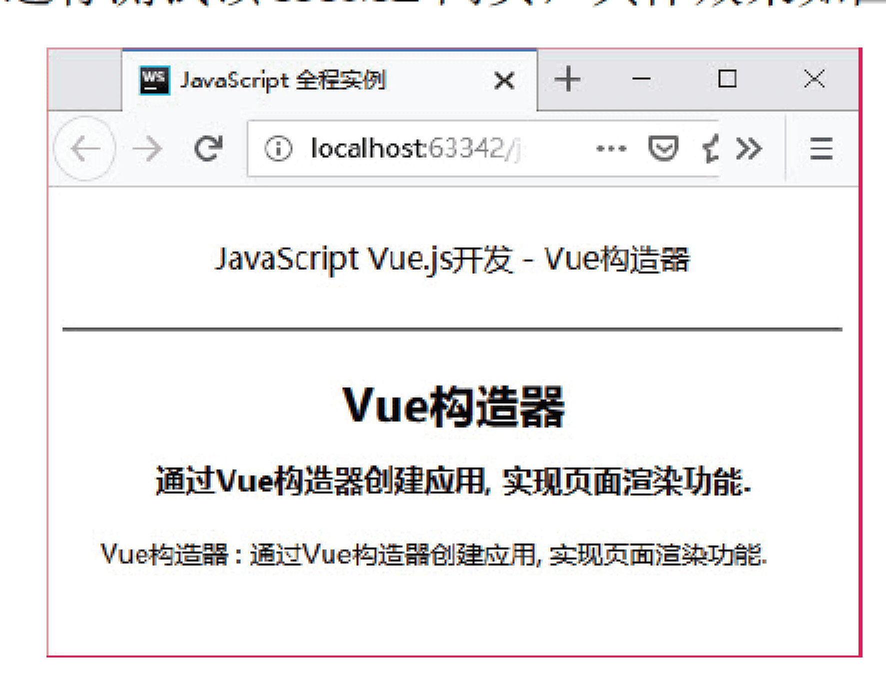


图 13.3　Vue.js 构造器

如图 13.3 所示，页面中成功显示了通过 Vue.js 框架渲染出的文本内容（包括通过函数 all()所组合的内容）。

13.4　Vue.js 构造器属性修改

我们再回过头来看看【代码 13-2】：Vue 构造器中“data”参数所定义的属性其实就是一个 JSON 结构数据，那么是不是可以单独定义这个属性呢？下面先看一个关于使用 Vue 构造器“data”参数定义属性的代码实例。

【代码 13-3】（详见源代码目录 ch13-vue-js-vue-data.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <script src="js/vue.js"></script>
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript Vue.js 开发 - Vue 构造器"data"参数</nav>
</header>
<div id='id-div-vue'>
    <h1>{{ title }}</h1>
    <h3>{{ content }}</h3>
    <p>{{ all() }}</p>
</div>
<script type="text/javascript">
    var data = {
        title : "Vue 构造器\"data\"参数",
        content : "定义\"data\"参数属性的另一种方法."
    }
    var app = new Vue({
        el: '#id-div-vue',
        data: data,
        methods: {
            all : function () {
                return this.title + " : " + this.content;
            }
        }
    });
```

```
31 </script>
32 </body>
33 </html>
```

关于【代码 13-3】的说明：

- 【代码 13-3】是在【代码 13-2】的基础上修改而成的，主要就是改变了“data”参数属性值的定义方式。
- 第 18～21 行代码定义了一个 JSON 结构数据对象（data）。注意，这里使用了与 Vue 构造器“data”参数同名的形式，是为了后面的内容做一个铺垫。
- 第 24 行代码通过“data”参数定义了属性（data），该属性（data）就是第 18～21 行代码定义的数据对象。
- 第 25～29 行代码通过“methods”参数定义了一个函数（all），特别之处是仍使用了属性（title 和 content）。读者可能会有疑问，第 24 行代码中并没有显式地定义这两个属性（title 和 content），代码会不会报错呢？没关系，我们通过下面的测试结果看一下。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 13.4 所示。

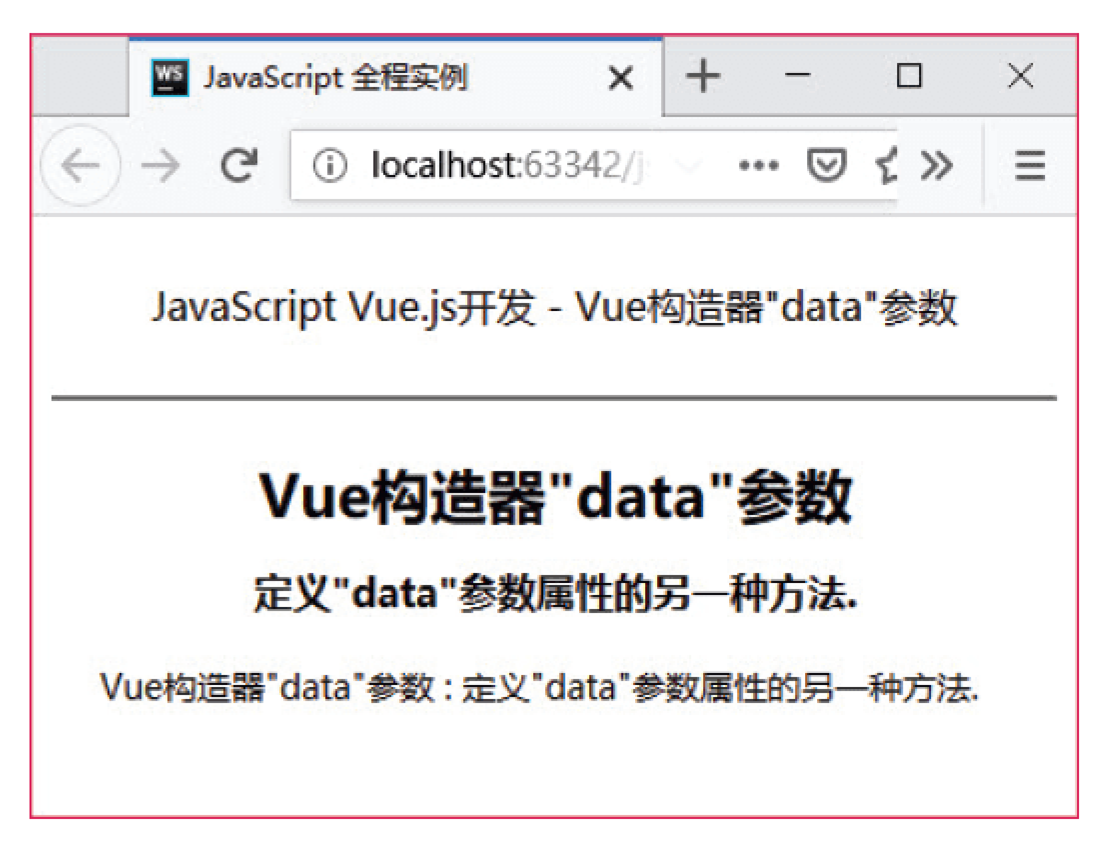


图 13.4 Vue.js 构造器"data"参数

如图 13.4 所示，页面中成功显示了通过 Vue.js 框架渲染出的文本内容，包括通过 JSON 结构数据对象（data）定义的属性以及函数方法（all）对属性（title 和 content）的引用的代码都正确执行了。

为了进一步验证 Vue.js 构造器所定义的属性与页面视图中所渲染的内容之间完全是响应式的，可以通过修改属性的操作进行测试。下面看一个通过 Vue 构造器修改属性实现响应式页面效果的代码实例。

【代码 13-4】（详见源代码目录 ch13-vue-js-vue-modify-data.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <script src="js/vue.js"></script>
```

```
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript Vue.js 开发 - Vue 构造器修改属性</nav>
</header>
<div id='id-div-vue'>
    <h1>{{ title }}</h1>
    <h3>{{ content }}</h3>
    <p>{{ all() }}</p>
</div>
<div>
    <button id="id-btn-data" onclick="on_btn_data()">通过 data 对象修改属性
                                                        </button>
    <button id="id-btn-app" onclick="on_btn_app()">通过 app 对象修改属性
                                                        </button>
</div>
<script type="text/javascript">
    var data = {
        title : "Vue 构造器\"data\"参数",
        content : "定义\"data\"参数属性的另一种方法."
    }
    var app = new Vue({
        el: '#id-div-vue',
        data: data,
        methods: {
            all : function () {
                return this.title + " : " + this.content;
            }
        }
    });
    /**
     * func button click - data
     */
    function on_btn_data() {
        if(app.title === data.title) {
            data.title = "Vue 构造器修改属性";
        }
    }
    /**
     * func button click - app
     */
```

```
46     function on_btn_app() {
47         if(app.content === data.content) {
48             app.content = "通过 app 对象修改 content 属性.";
49         }
50     }
51 </script>
52 </body>
53 </html>
```

关于【代码 13-4】的说明：

- 【代码 13-4】是在【代码 13-3】的基础上修改而成的，主要就是增加了两个按钮，用于实现修改属性的操作。
- 第 38～42 行代码实现了第 18 行代码所定义<button>按钮（id="id-btn-data"）的单击 onclick 事件方法。第 39 行代码先判断分别通过 Vue 对象（app）与 JSON 对象（data）引用的属性（title）是否恒等，然后第 40 行代码通过 JSON 对象（data）修改属性（title）值。
- 第 46～50 行代码与第 38～42 行代码类似，不同之处是第 48 行代码通过 Vue 对象（app）修改属性（content）值。

下面使用 Firefox 浏览器运行测试该 HTML 网页，初始效果如图 13.5 所示。

在图 13.5 中，我们单击页面中的“通过 data 对象修改属性”按钮，页面效果如图 13.6 所示。

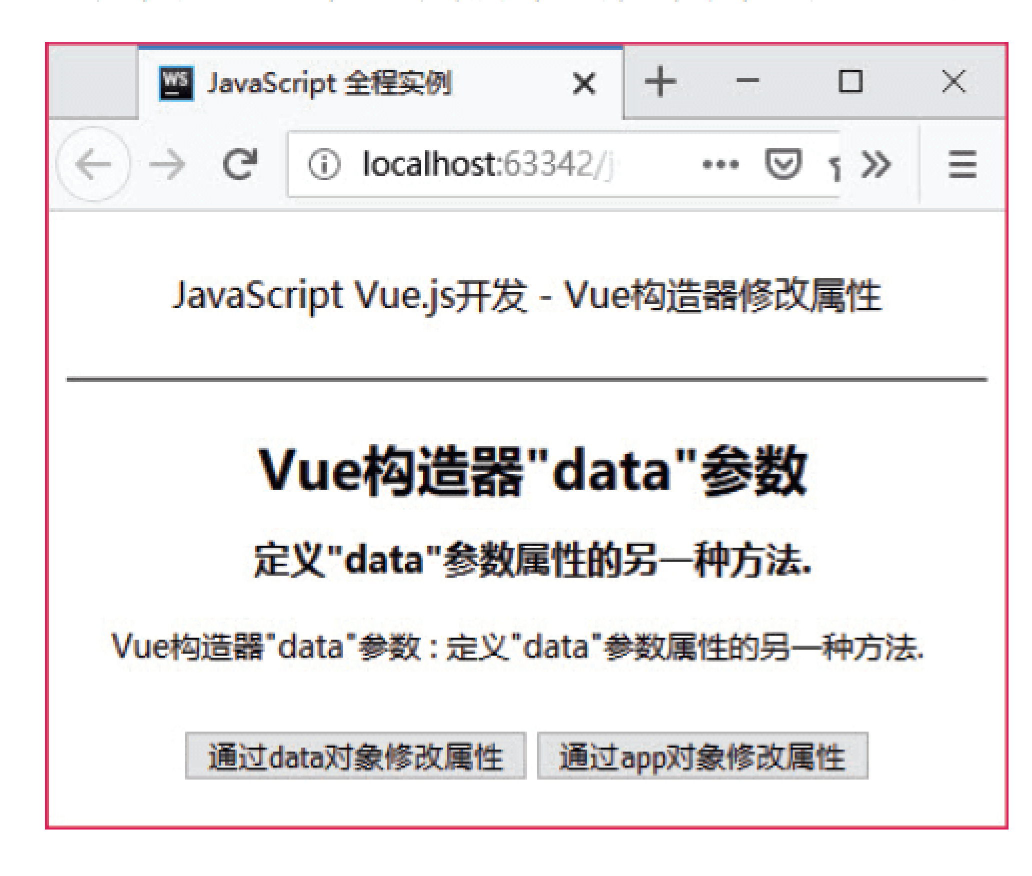


图 13.5　Vue.js 构造器修改属性（一）

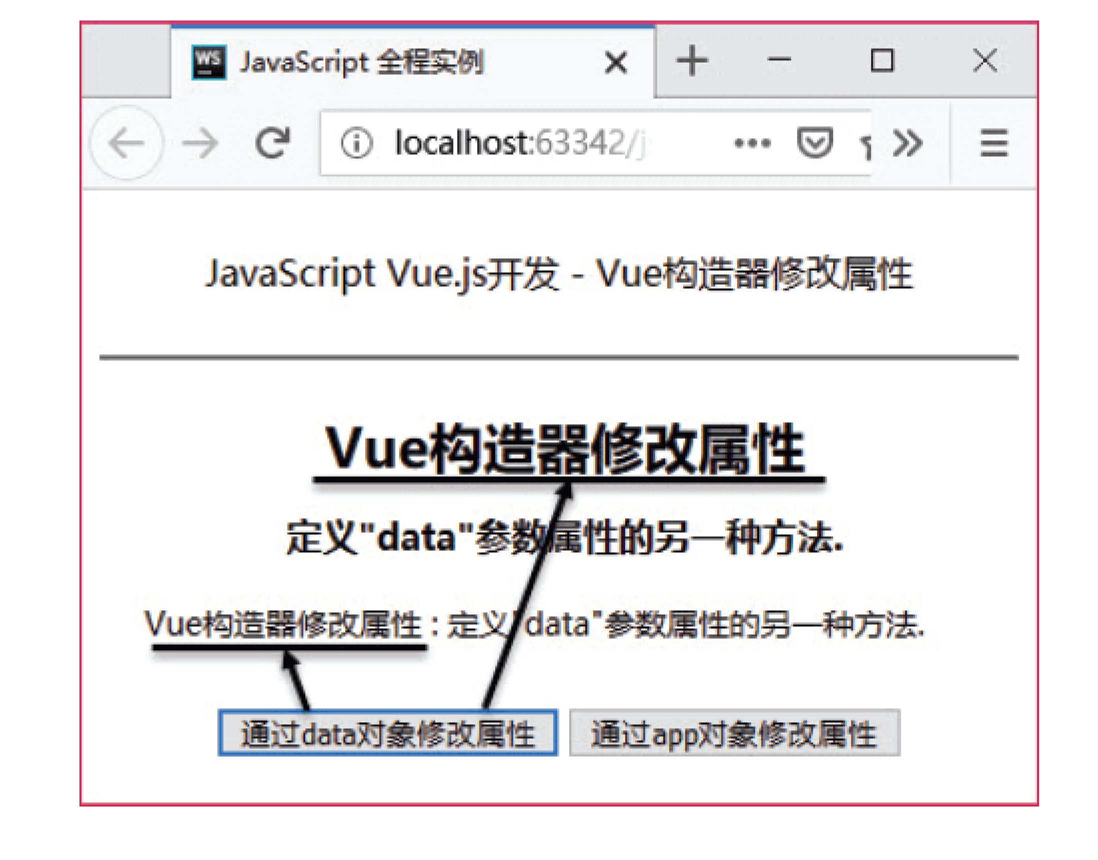


图 13.6　Vue.js 构造器修改属性（二）

如图 13.6 中的箭头所示，通过 JSON 对象（data）修改的属性（title）值成功渲染到页面中了。下面我们接着单击页面中的“通过 app 对象修改属性”按钮，页面效果如图 13.7 所示。

如图 13.7 中的箭头所示，通过 Vue 对象（app）修改的属性（content）值同样成功渲染到页面中了。

读者可能注意到了，在【代码 13-4】中分别通过 Vue 对象（app）和 JSON 对象（data）修改了两个属性值（title 和 content），那么通过一个对象修改的属性值是不是同步到通过另一个对象来引用相同的属性值呢？答案是肯定的。

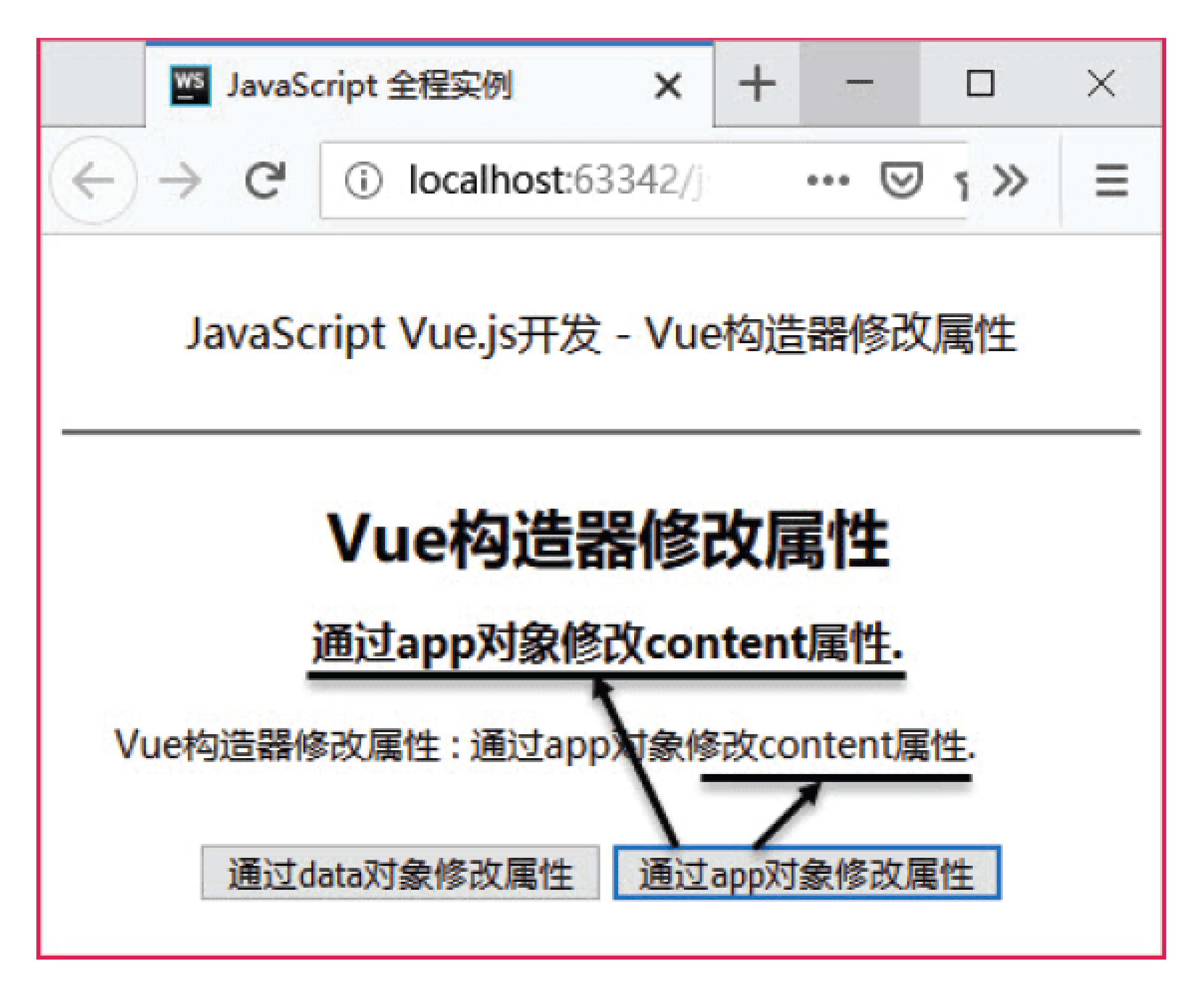


图 13.7　Vue.js 构造器修改属性（三）

为了验证这个同步性，我们可以在【代码 13-4】中加入浏览器控制台调试代码进行属性值的同步输出，具体代码如下。

【代码 13-5】（详见源代码目录 ch13-vue-js-vue-data-sync.html 文件）

```
<script type="text/javascript">
    /**
     * func button click - data
     */
    function on_btn_data() {
        if(app.title === data.title) {
            data.title = "Vue 构造器修改属性";
            console.log("修改 data.title 为: Vue 构造器修改属性");
            console.log("app.title= " + app.title);
        }
    }
    /**
     * func button click - app
     */
    function on_btn_app() {
        if(app.content === data.content) {
            app.content = "通过 app 对象修改 content 属性.";
            console.log("修改 app.content 为: 通过 app 对象修改 content 属性.");
            console.log("data.content= " + data.content);
        }
    }
</script>
```

关于【代码 13-5】的说明：

- 【代码 13-5】是在【代码 13-4】的基础上修改而成的，主要就是增加了第 09 行代码和第 19 行代码，通过浏览器控制台进行属性值的同步输出验证。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 13.8 和图 13.9 所示。

如图 13.8 中的箭头和标识所示，通过 JSON 对象（data）修改的属性（title）值会同步到“app.title”属性中。

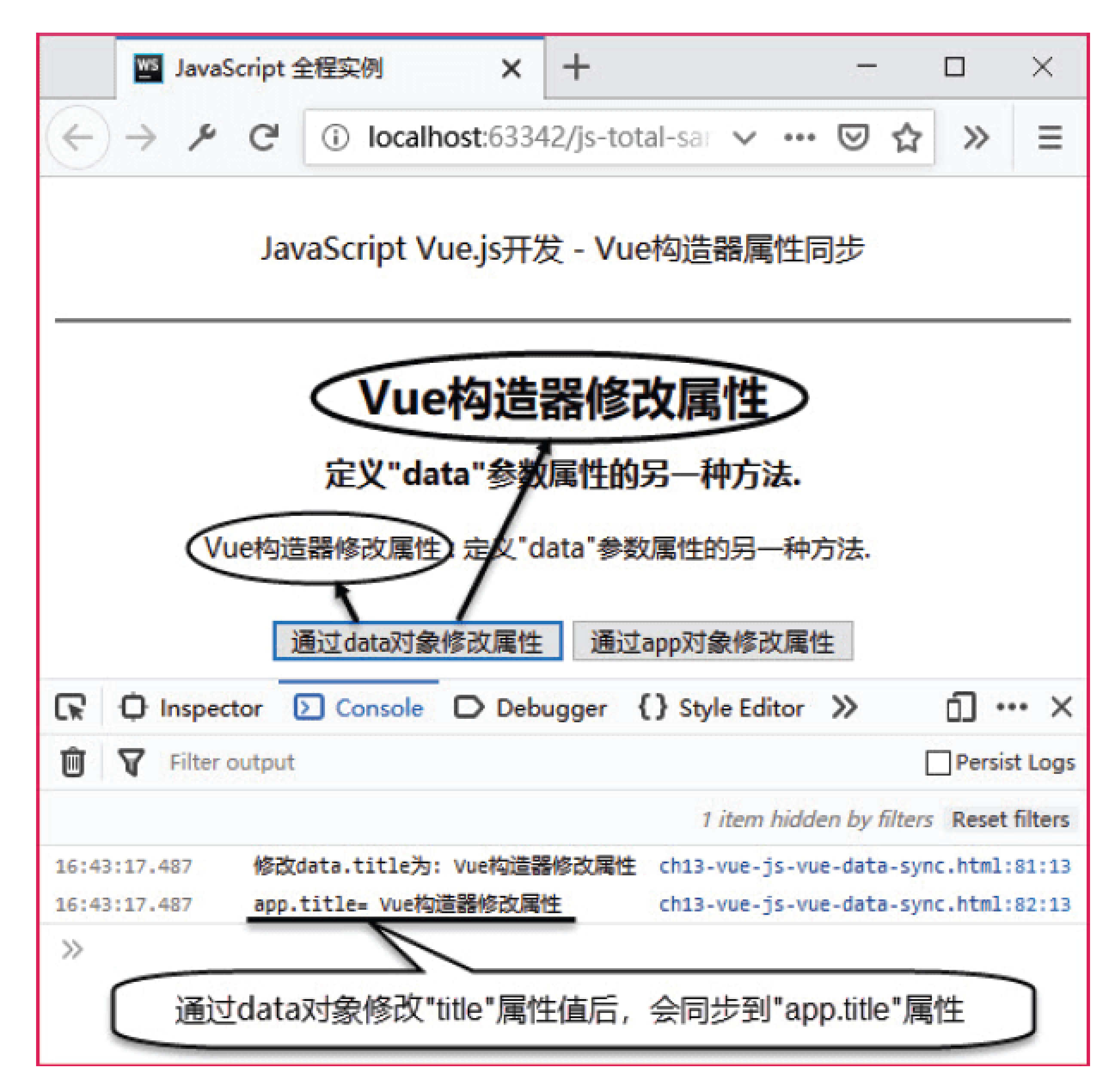


图 13.8　Vue.js 构造器属性同步（一）

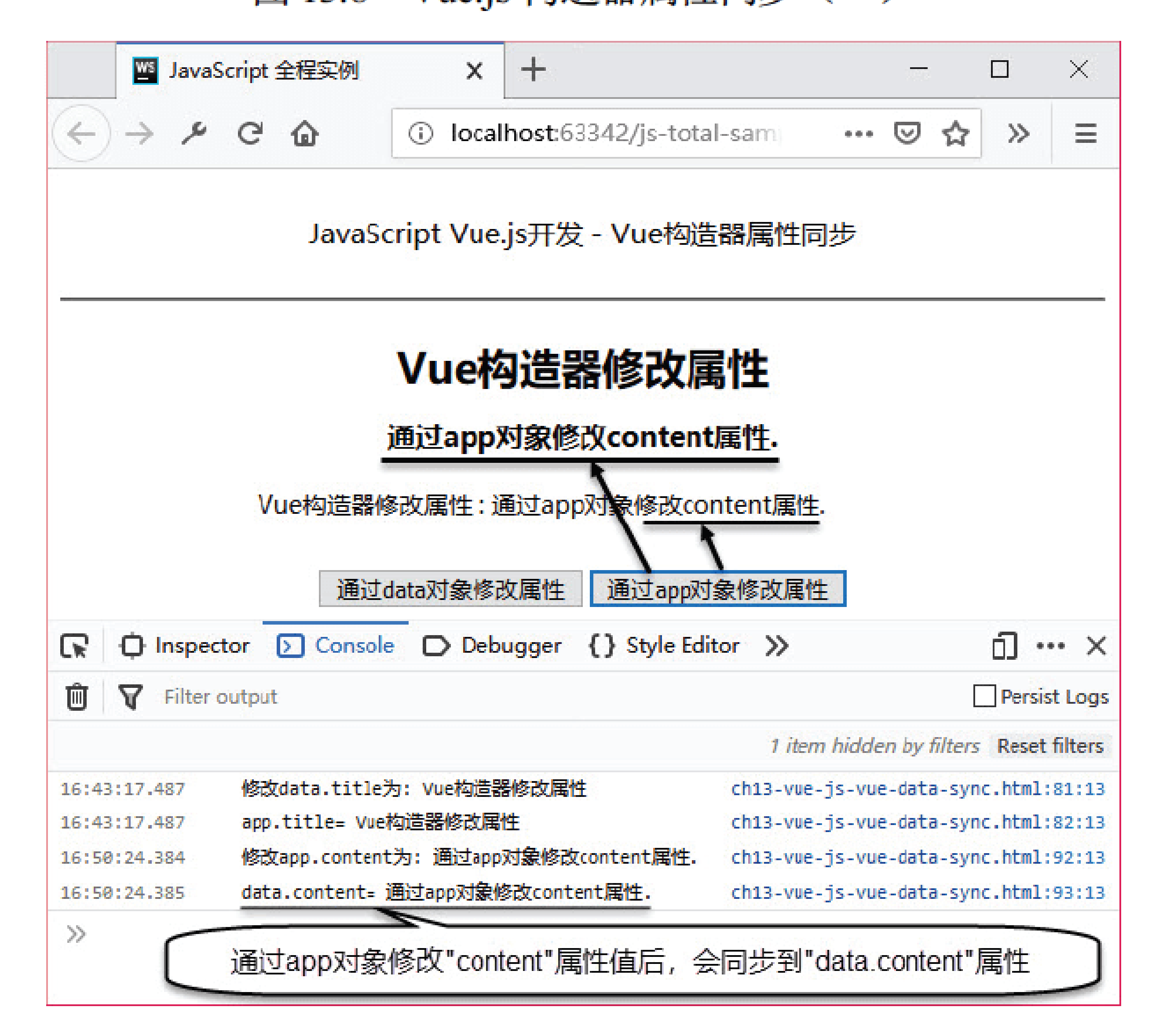


图 13.9　Vue.js 构造器属性同步（二）

如图 13.9 中的箭头和标识所示，通过 Vue 对象（app）修改的属性（content）值会同步到“data.content”属性中。

13.5　Vue.js 构造器参数引用

在 Vue.js 框架中，还提供了一个“$”前缀符号来引用 Vue 构造器的参数，以便将其与用户自定义的属性进行区分。下面看一个关于使用 Vue 构造器的“$”前缀符号引用参数的代码实例。

【代码 13-6】（详见源代码目录 ch13-vue-js-vue-params.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <script src="js/vue.js"></script>
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript Vue.js 开发 - Vue 构造器参数引用</nav>
</header>
<div id='id-div-vue'>
    <h1>{{ title }}</h1>
    <h3>{{ content }}</h3>
    <p>{{ getData() }}</p>
    <p>{{ getEl() }}</p>
</div>
<script type="text/javascript">
    var data = {
        title : "Vue 构造器参数引用",
        content : "通过 Vue 构造器参数渲染页面."
    }
    var app = new Vue({
        el: '#id-div-vue',
        data: data,
        methods: {
            getData : function () {
                if(this.$data === data) {
                    return eval(this.$data);
                }
            }
```

```
32              getEl : function() {
33                  if(this.$el === document.getElementById('id-div-vue')) {
34                      return this.$el.innerText;
35                  }
36              }
37          }
38      });
39  </script>
40  </body>
41  </html>
```

关于【代码 13-6】的说明：

- 第 27 ~ 31 行代码在“methods”参数中定义了第一个函数方法（getData），具体内容如下：
    - 第 28 行代码先通过 if 条件语句判断 Vue 构造器（app）的 data 参数（注意需要通过“$”符号引用）与“data”属性（指代第 19 ~ 22 行代码定义的 JSON 对象）是否恒等；
    - 如果判断结果为“true”，第 29 行代码会直接返回 data 参数（this.$data）。
- 第 32 ~ 36 行代码在“methods”参数中定义了第二个函数方法（getEl），具体内容如下：
    - 第 33 行代码先通过 if 条件语句判断 Vue 构造器（app）的 el 参数（注意需要通过“$”符号引用）与 DOM 对象（<div id='id-div-vue'>）是否恒等；
    - 如果判断结果为“true”，第 34 行代码会直接返回 DOM 对象（this.$el）的文本内容（innerText）。

下面使用 Firefox 浏览器运行测试该 HTML 网页，具体效果如图 13.10 所示。



图 13.10　Vue.js 构造器参数引用

13.6　Vue.js 模板语法

Vue.js 框架使用了基于 HTML 的模板语法，其核心思想就是允许设计人员使用简洁的模板语法以声明的方式将数据渲染进 DOM 对象中。同时，借助 Vue.js 框架特有的响应方式，即在应用状态发生改变时能够优化计算出重新渲染组件的最小代价并应用到 DOM 对象中。

其实，在前面几个代码实例中所使用的“{{　}}”就是最简单的文本模板插入方式。当然，除了最基础的文本插入方式之外，还有 HTML 代码插入、属性插入、表达式插入等。在本节中，我们简要介绍一下关于 Vue.js 框架模板语法的内容。

首先，我们看一个比较特殊的、关于使用文本模板插入方式的代码示例。

【代码 13-7】（详见源代码目录 ch13-vue-js-tmpl-once.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <script src="js/vue.js"></script>
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript Vue.js 开发 - Vue 模板语法(文本插入)</nav>
</header>
<div id='id-div-vue'>
    <h3>{{ title }}</h3>
    <p v-text="content"></p>
    <p v-once>{{ contentOnce }}</p>
</div>
<div>
  <button id="id-btn-change-data" onclick=
                              "on_btn_change_text()">尝试修改文本</button>
</div>
<script type="text/javascript">
    /**
     * Vue constructor
     */
    var app = new Vue({
        el: '#id-div-vue',
        data: {
            title : "Vue 模板语法(文本插入)",
```

```
28              content : "尝试修改文本行的文本内容.",
29              contentOnce : "尝试修改本行的文本内容."
30          }
31      });
32      /**
33       * func button click - data
34       */
35      function on_btn_change_text() {
36          app.$data.content = "修改本行的文本内容完成!";
37          console.log("data.content is modified : " + app.$data.content);
38          app.$data.contentOnce = "修改本行的文本内容完成!";
39          console.log("data.contentOnce is modified : " +
                                                    app.$data.contentOnce);
40      }
41  </script>
42  </body>
43  </html>
```

关于【代码 13-7】的说明：

- 第 14 行代码在<p>标签元素内通过“v-”前缀指令（v-text）定义了一个插入文本（content），对应第 28 行代码定义的 Vue 构造器属性（content）值。
- 第 15 行代码在<p>标签元素内添加了一个“v-”前缀指令（v-once），该指令将会执行一次性插入文本的操作。具体来讲，就是当改变第 29 行定义的 Vue 构造器属性（contentOnce）值时，不会同步更新第 15 行代码中属性（contentOnce）值所显示的内容。
- 第 35～40 行代码实现的按钮单击事件方法（on_btn_change_text()）同时修改了 Vue 构造器属性 content 和 contentOnce 的值，并在浏览器控制台中输出了修改后的结果。

下面使用 Firefox 浏览器运行测试该 HTML 网页，初始效果如图 13.11 所示。

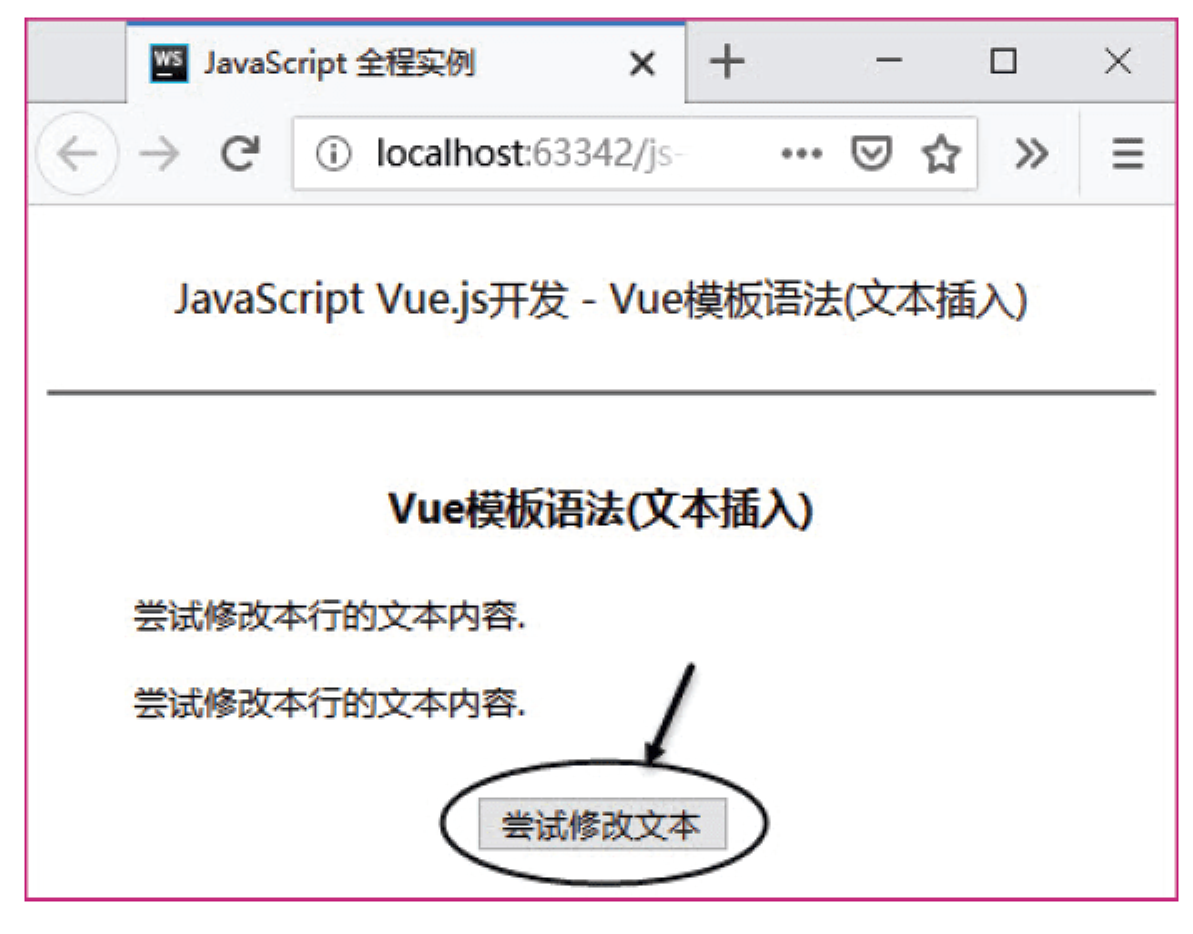


图 13.11　Vue.js 模板语法之文本插入（一）

如图 13.11 中的箭头所示，单击“尝试修改文本”按钮，页面效果如图 13.12 所示。

图 13.12　Vue.js 模板语法之文本插入（二）

如图 13.12 中的箭头和标识所示，单击“尝试修改文本”按钮后，根据浏览器控制台中的输出，可以看出两个 Vue 构造器属性（content 和 contentOnce）值已经更改了。同时，第 14 行代码定义的插入文本（content）进行了同步更新，但第 15 行代码由于添加了“v-”前缀指令（v-once）而没有得到同步更新。

Vue.js 框架模板不仅仅支持文本插入（v-text），还支持直接插入 HTML 代码进行渲染，具体方法是通过“v-”前缀指令（v-html）来实现的。下面我们看一个关于使用 HTML 代码模板插入方式的代码示例。

【代码 13-8】（详见源代码目录 ch13-vue-js-tmpl-html.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <script src="js/vue.js"></script>
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript Vue.js 开发 - Vue 模板语法(HTML 插入)</nav>
</header>
<div id='id-div-vue' v-html="htmlContent">
```

```
13 </div>
14 <script type="text/javascript">
15     /**
16      * Vue constructor
17      */
18     var app = new Vue({
19         el: '#id-div-vue',
20         data: {
21     htmlContent : '<h3>Vue 模板语法(HTML 插入)</h3><p>
                                                 使用 HTML 代码插入方式.</p>'
22         }
23     });
24 </script>
25 </body>
26 </html>
```

关于【代码 13-8】的说明：

- 第 12 行代码在<div>标签元素内通过“v-”前缀指令（v-html）定义了一段 HTML 插入代码，对应第 21 行代码定义的 Vue 构造器属性 htmlContent 值。
- 在 Vue 构造器内的第 21 行代码中，通过属性 htmlContent 定义了一段 HTML 代码（<h3>、<p>）。

下面使用 Firefox 浏览器运行测试该 HTML 网页，初始效果如图 13.13 所示。

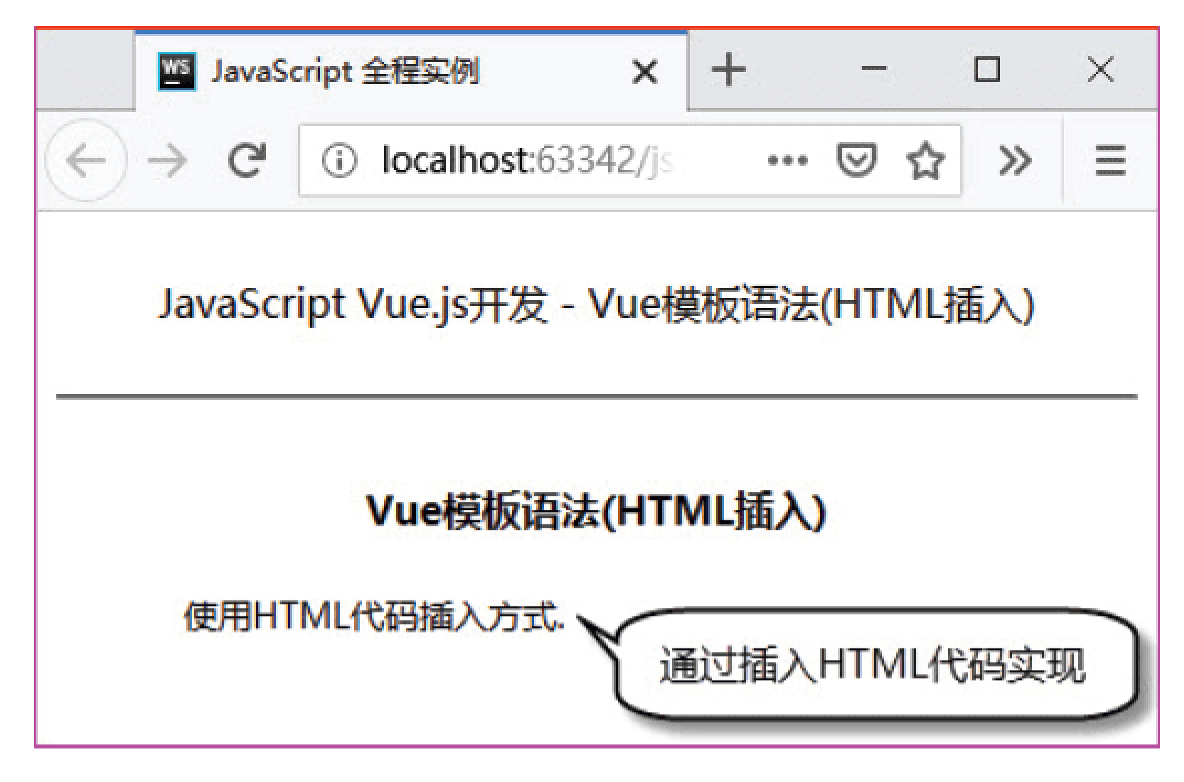


图 13.13　Vue.js 模板语法之 HTML 插入

Vue.js 框架还设计了一个非常有意思的动态绑定功能，通过“v-”前缀指令（v-bind）来实现。举个例子说明一下，我们知道 HTML 5 标准规范中为标签元素新增了一个“title”属性，可以实现“当鼠标停留在该元素上时显示出一个工具提示文本（Tooltip Text）”的效果。借助“v-”前缀指令（v-bind），可以实现功能更强的动态绑定效果。下面看一个具体的代码示例。

【代码 13-9】（详见源代码目录 ch13-vue-js-tmpl-bind.html 文件）

```
01 <!DOCTYPE html>
02 <html>
03 <head>
04     <script src="js/vue.js"></script>
05     <title>JavaScript 全程实例</title>
06 </head>
07 <body>
08 <!-- 添加文档主体内容 -->
09 <header>
10     <nav>JavaScript Vue.js 开发 - Vue 模板语法(动态绑定)</nav>
11 </header>
12 <div id='id-div-vue'>
13     <h3 v-text="title"></h3>
14     <p v-bind:title="htmlContent">页面更新时间(通过鼠标悬停查看)</p>
15 </div>
16 <script type="text/javascript">
17     /**
18      * Vue constructor
19      */
20     var app = new Vue({
21        el: '#id-div-vue',
22        data: {
23            title : "动态绑定 HTML 代码",
24            htmlContent : new Date().toLocaleString()
25        }
26     });
27 </script>
28 </body>
29 </html>
```

关于【代码 13-9】的说明：

- 第 14 行代码在<p>标签元素内通过“v-”前缀指令（v-bind）绑定了“title”属性（代码形式为 v-bind:title），绑定的内容是第 24 行代码定义的 Vue 构造器属性 htmlContent 值。
- 在 Vue 构造器内的第 24 行代码中，通过属性 htmlContent 定义了一段 JavaScript 代码（通过 Date 对象获取了当前时间）。

下面使用 Firefox 浏览器运行测试该 HTML 网页，初始效果如图 13.14 所示。

如图 13.14 中的箭头所示，当我们将鼠标悬停在提示文本上时会弹出一个工具提示文本（Tooltip Text），显示当前时间。

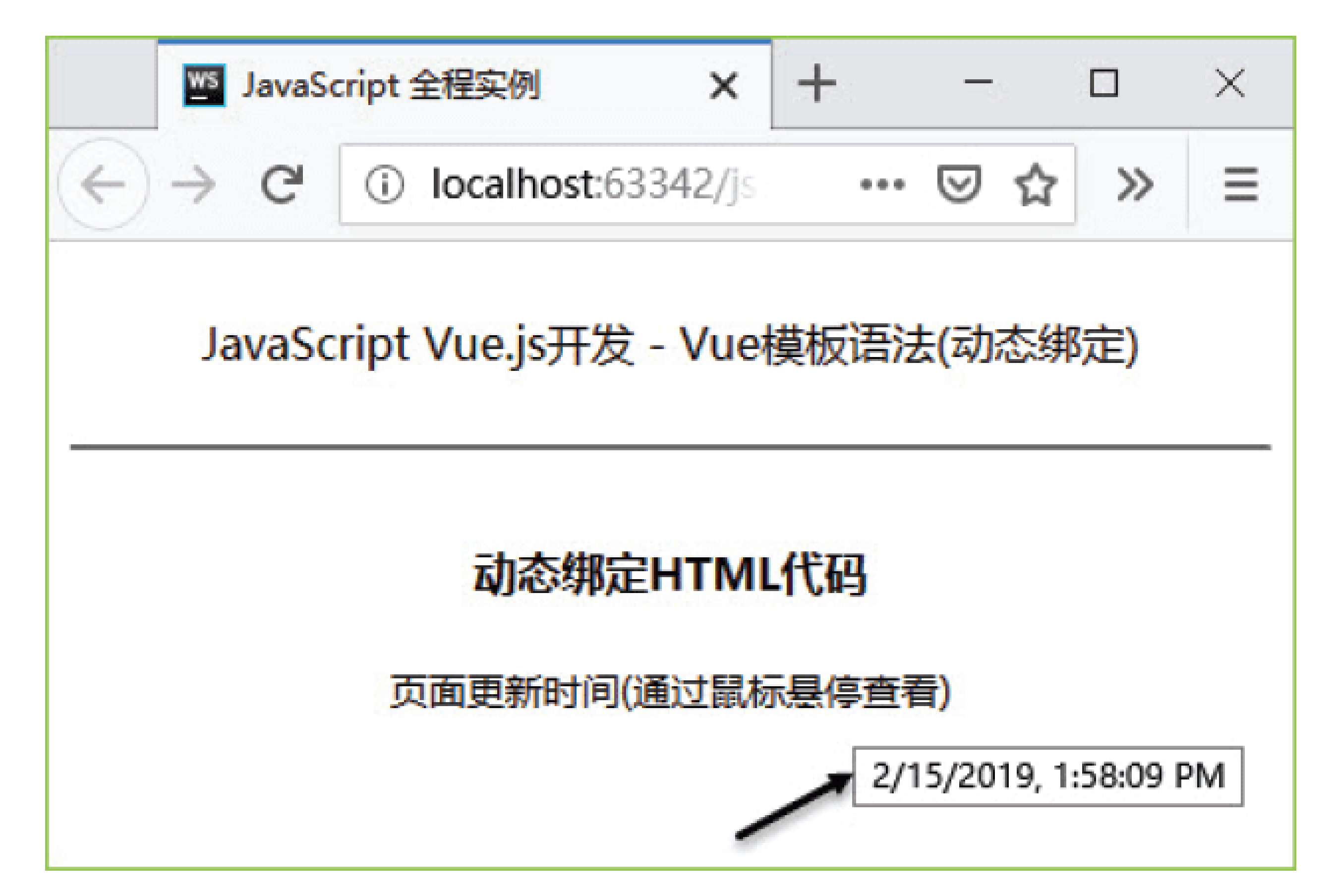


图 13.14　Vue.js 模板语法之动态绑定

13.7　Vue.js 条件循环语句

Vue.js 框架设计的“v-”前缀指令非常强大，提供了带有逻辑控制功能的条件语句（v-if）和循环语句（v-for）。比如，使用 Vue 条件语句就可以很方便地实现“显示或隐藏”元素内容的功能。在传统的 JavaScript 方式下，需要编写相对复杂的切换 CSS 样式的代码来实现。

首先，我们看一个使用 Vue 条件语句控制页面元素内容切换“显示/隐藏”效果的代码实例。

【代码 13-10】（详见源代码目录 ch13-vue-js-v-if.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <script src="js/vue.js"></script>
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript Vue.js 开发 - Vue 条件语句</nav>
</header>
<div id='id-div-vue'>
    <h1>{{ title }}</h1>
    <p v-if="show">{{ content }}</p>
    <p v-if="hide">{{ content }}</p>
</div>
<script type="text/javascript">
    var app = new Vue({
        el: '#id-div-vue',
```

```
20          data: {
21              title: "Vue 条件语句",
22              content: "文本：通过 Vue 条件语句可以实现可见/隐藏效果.",
23              show: true,
24              hide: false
25          }
26      });
27  </script>
28  </body>
29  </html>
```

关于【代码 13-10】的说明：

- 第 14 行和第 15 行代码分别在<p>标签元素内通过“v-”前缀指令（v-if）定义了一个条件语句，分别对应第 23 行和第 24 行代码定义的 Vue 构造器属性（show:true 和 hide:false）值。
- 在 Vue 构造器内，第 23 行代码中的属性（show）定义为 true（真）值，而第 24 行代码中的属性（hide）定义为 false（假）值。

下面使用 Firefox 浏览器运行测试该 HTML 网页，初始效果如图 13.15 所示。

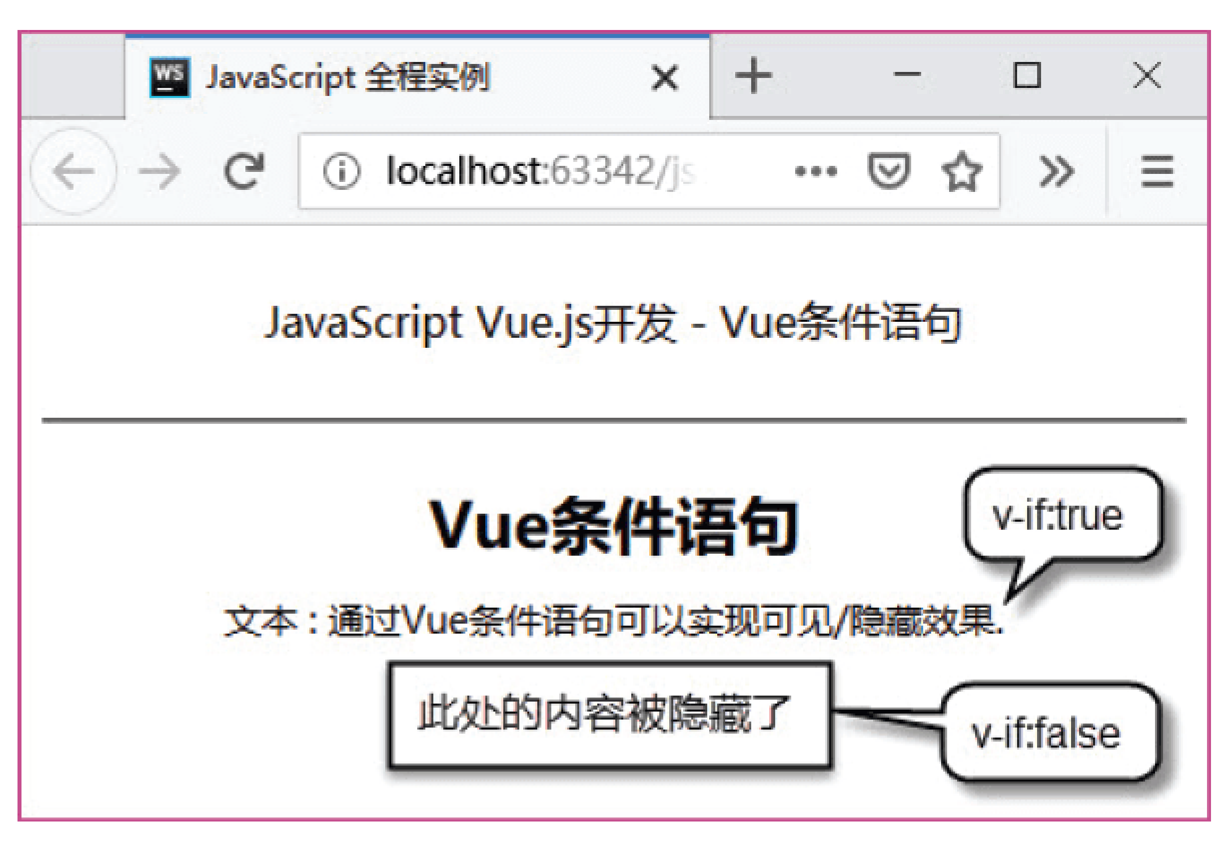


图 13.15　Vue.js 条件语句

如图 13.15 中的箭头和标识所示，通过 Vue 条件语句（v-if:true）插入的段落内容可以显示出来，而通过 Vue 条件语句（v-if:false）插入的段落内容则被隐藏了。

下面我们看一个使用 Vue 循环语句在页面中控制输出列表的代码实例。

【代码 13-11】（详见源代码目录 ch13-vue-js-v-for.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <script src="js/vue.js"></script>
    <title>JavaScript 全程实例</title>
</head>
```

```
07 <body>
08 <!-- 添加文档主体内容 -->
09 <header>
10     <nav>JavaScript Vue.js 开发 - Vue 循环语句</nav>
11 </header>
12 <div id='id-div-vue'>
13     <h1>{{ title }}</h1>
14     <ol>
15         <li v-for="li in arrLi">
16             {{ li.text }}
17         </li>
18     </ol>
19 </div>
20 <script type="text/javascript">
21     var app = new Vue({
22         el: '#id-div-vue',
23         data: {
24             title: "循环语句",
25             arrLi: [
26                 {text: 'JavaScript'},
27                 {text: 'React'},
28                 {text: 'Vue.js'}
29             ]
30         }
31     });
32 </script>
33 </body>
34 </html>
```

关于【代码 13-11】的说明：

- 第 14 ~ 18 行代码通过
标签元素定义了一个列表，具体说明如下：
  - 第 15 行代码的<li>标签元素内通过“v-”前缀指令(v-for)定义了一个循环语句(li in arrLi)。相信读者发现了，这里很像传统的“for…in…”循环语句。确实如此，“li”本身是一个自变量，“arrLi”对应第 25 ~ 29 行代码定义的 Vue 构造器属性（arrLi）数组值。
  - 第 16 行代码通过自变量 li（“arrLi”数组项）引用“text”属性，实现列表项的输出。
- 在 Vue 构造器内，第 25 ~ 29 行代码定义的属性（arrLi）是一个数组结构，这个数组其实就是一个 JSON 结构的对象。

下面使用 Firefox 浏览器运行测试该 HTML 网页，初始效果如图 13.16 所示。



图 13.16 Vue.js 循环语句

13.8 Vue.js 事件监听处理

Vue.js 是一个基于 JavaScript 实现的高性能框架，事件监听处理功能是必须具备的。Vue.js 事件监听是通过“v-”前缀指令（v-on）实现的，使用“on”是继承自 JavaScript 事件方法的名称（比如 onclick、onchange、onfocus 等）。

在前文的 Vue 条件语句中，我们实现了“显示/隐藏”内容的效果。下面我们在此基础上实现一个通过用户操作切换内容“显示/隐藏”效果的实例。

【代码 13-12】（详见源代码目录 ch13-vue-js-v-on.html 文件）

```
<!DOCTYPE html>
<html>
<head>
    <script src="js/vue.js"></script>
    <title>JavaScript 全程实例</title>
</head>
<body>
<!-- 添加文档主体内容 -->
<header>
    <nav>JavaScript Vue.js 开发 - Vue 事件监听处理</nav>
</header>
<div id='id-div-vue'>
    <h1>{{ title }}</h1>
    <p v-if="show" v-html="contentShow"></p><br>
    <p v-if="hide" v-html="contentHide"></p><br>
    <button v-on:click="onShowHide">切换“显示/隐藏”效果</button>
</div>
```

```
18 <script type="text/javascript">
19     /**
20      * Vue constructor
21      */
22     var app = new Vue({
23         el: '#id-div-vue',
24         data: {
25             title: "Vue 条件语句",
26             contentShow: "通过 Vue 条件语句可以实现<b>可见</b>效果.",
27             contentHide: "通过 Vue 条件语句可以实现<b>隐藏</b>效果.",
28             show: true,
29             hide: false
30         },
31         methods: {
32             /**
33              * func - change show/hide effect
34              */
35             onShowHide: function () {
36                 if(this.show) {
37                     console.log(this.show);
38                     this.show = false;
39                     this.hide = true;
40                 } else {
41                     console.log(this.show);
42                     this.show = true;
43                     this.hide = false;
44                 }
45             }
46         }
47     });
48 </script>
49 </body>
50 </html>
```

关于【代码 13-12】的说明：

- 在第 14 行和第 15 行代码中，分别在<p>标签元素内通过“v-”前缀指令（v-if）定义了一个条件语句，分别对应第 28 行和第 29 行代码定义的 Vue 构造器属性（show:true 和 hide:false）值；同时，在<p>标签元素内还通过“v-”前缀指令（v-html）定义了一段 HTML 插入代码，分别对应第 26 行和第 27 行代码定义的 Vue 构造器属性（contentShow 和 contentHide）值，我们的目标就是通过用户操作来切换显示这两个段落的内容。
- 在 Vue 构造器内，第 28 行代码中的属性（show）定义为 true（真）值，而第 29 行代码中的属性（hide）定义为 false（假）值。

- 在第 16 行代码中，在<button>标签元素内通过“v-”前缀指令（v-on:click）定义了一个单击事件处理方法（onShowHide）。
- 在第 31～46 行代码定义的“methods”参数中，第 35～45 行代码实现了单击事件处理方法（onShowHide）。第 36～44 行代码通过 if 条件语句切换定义属性（show）和属性（hide）的布尔值（true 或 false），从而实现响应式的内容“显示/隐藏”切换效果。

下面使用 Firefox 浏览器运行测试该 HTML 网页，初始效果如图 13.17 所示。

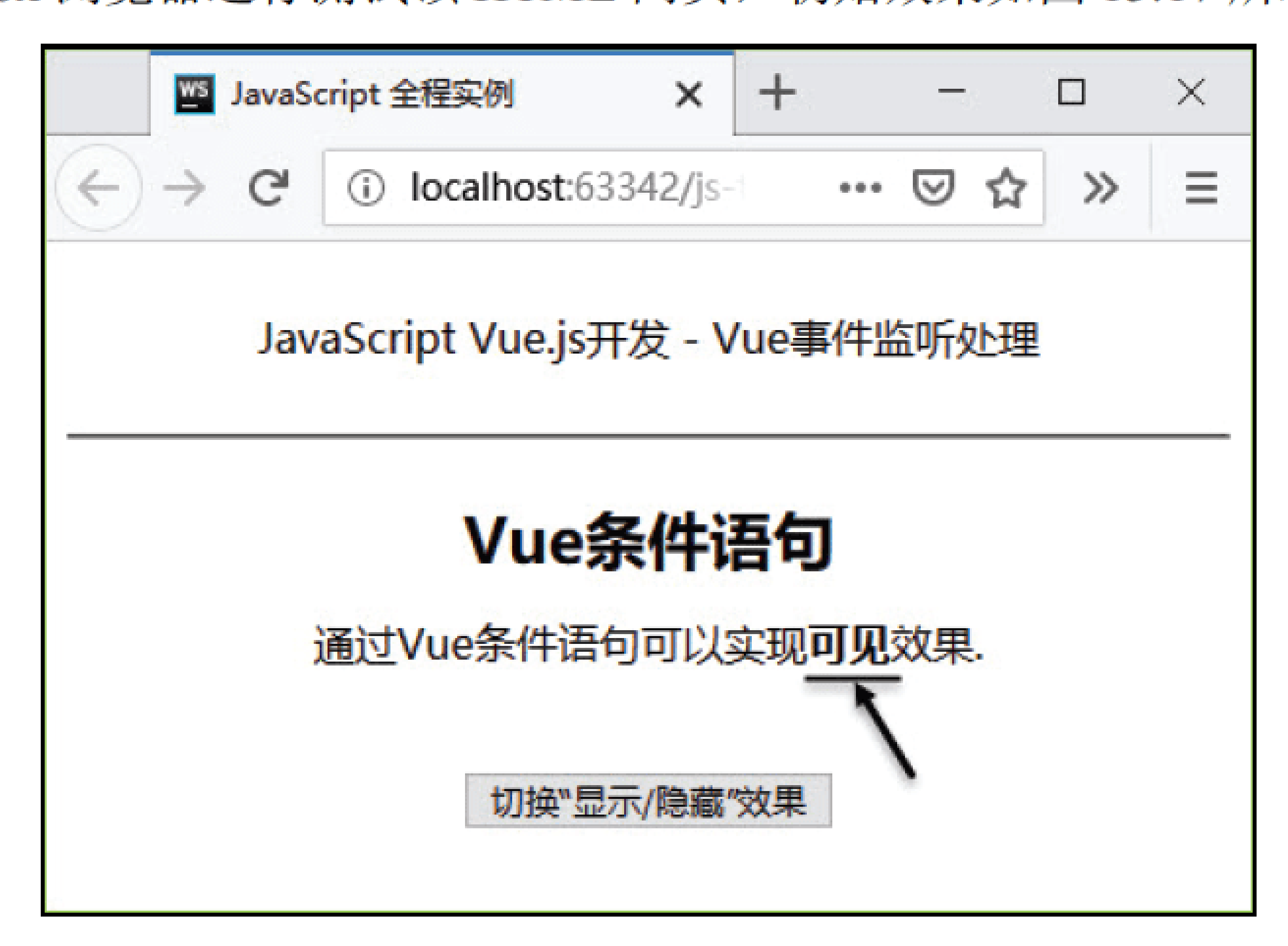


图 13.17　Vue.js 事件监听实现“显示/隐藏”切换效果（一）

如图 13.17 中的箭头和标识所示，页面初始状态显示的是“可见”效果的文本内容。下面，我们单击一次“切换"显示/隐藏"效果”按钮，页面更新效果如图 13.18 所示。

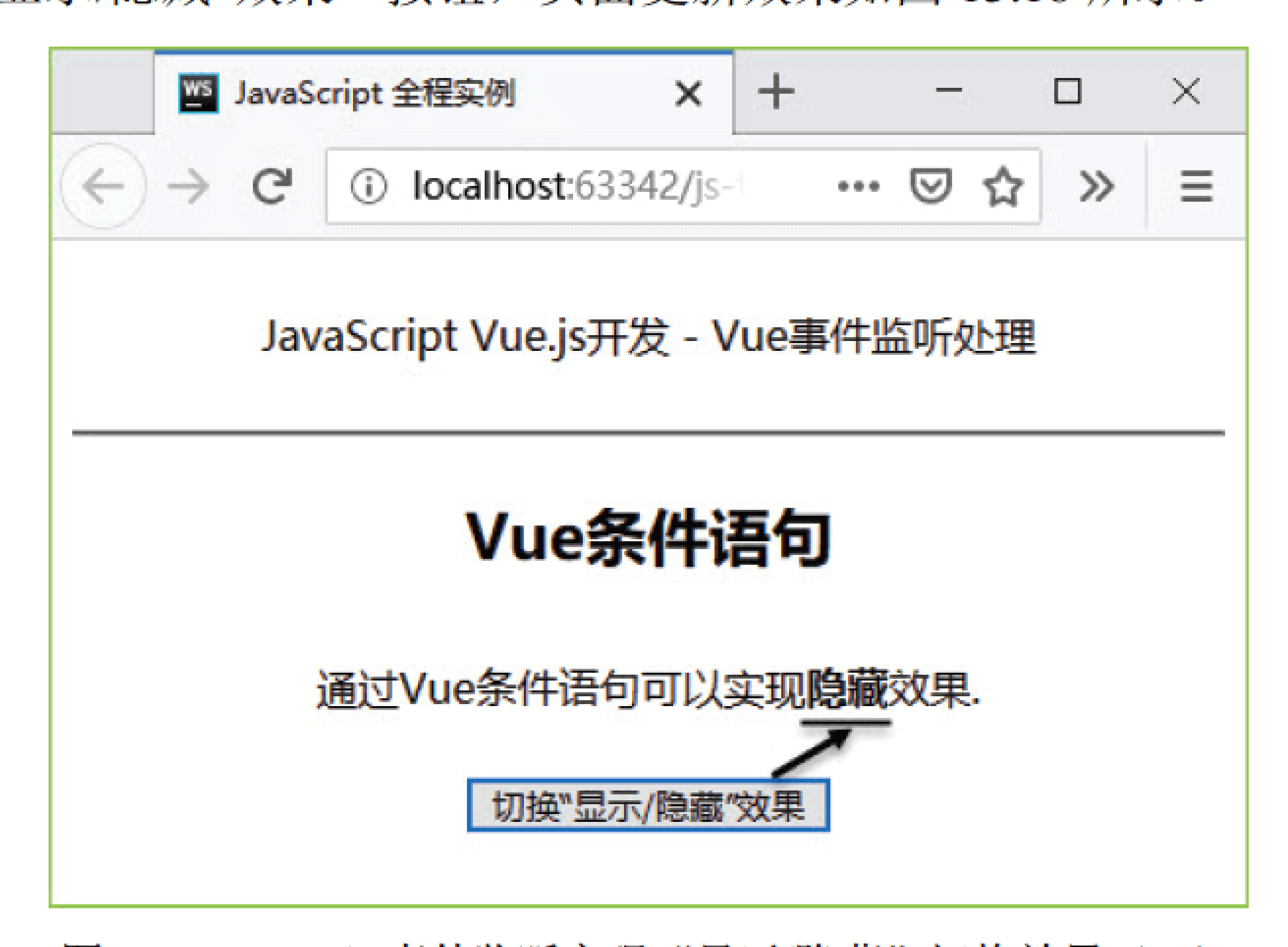


图 13.18　Vue.js 事件监听实现“显示/隐藏”切换效果（二）

如图 13.18 中的箭头和标识所示，此时页面状态切换到“隐藏”效果的文本内容。读者可以尝试连续单击“切换"显示/隐藏"效果”按钮，测试一下 Vue.js 框架特有的响应式的内容渲染切换效果。